

# Fast Arithmetic in Algorithmic Self-Assembly

Alexandra Keenan\*    Robert Schweller\*    Michael Sherman\*    Xingsi Zhong\*

## Abstract

In this paper we consider the time complexity of adding two  $n$ -bit numbers together within the tile self-assembly model. The (abstract) tile assembly model is a mathematical model of self-assembly in which system components are square tiles with different glue types assigned to tile edges. Assembly is driven by the attachment of singleton tiles to a growing seed assembly when the net force of glue attraction for a tile exceeds some fixed threshold. Within this framework, we examine the time complexity of computing the sum of 2  $n$ -bit numbers, where the input numbers are encoded in an initial seed assembly, and the output sum is encoded in the final, terminal assembly of the system. We show that this problem, along with multiplication, has a worst case lower bound of  $\Omega(\sqrt{n})$  in 2D assembly, and  $\Omega(\sqrt[3]{n})$  in 3D assembly. We further design algorithms for both 2D and 3D that meet this bound with worst case run times of  $O(\sqrt{n})$  and  $O(\sqrt[3]{n})$  respectively, which beats the previous best known upper bound of  $O(n)$ . Finally, we consider average case complexity of addition over uniformly distributed  $n$ -bit strings and show how to achieve  $O(\log n)$  average case time with a simultaneous  $O(\sqrt{n})$  worst case run time in 2D. As additional evidence for the speed of our algorithms, we implement our algorithms, along with the simpler  $O(n)$  time algorithm, into a probabilistic run-time simulator and compare the timing results.

## 1 Introduction.

Self-assembly is the process by which systems of simple objects autonomously organize themselves through local interactions into larger, more complex objects. Self-assembly processes are abundant in nature and serve as the basis for biological growth and replication. Understanding how to design and efficiently program molecular self-assembly systems promises to be fundamental for the future of nanotechnology. One particular direction of interest is the design of molecular computing systems for the efficient solution of fundamental computational problems. In this paper we study the complexity of computing arithmetic primitives within a well studied model of algorithmic self-assembly, the abstract tile assembly model.

The abstract tile assembly model (aTAM) models system monomers with four sided Wang tiles with glue types assigned to each edge. Assembly proceeds by tiles attaching, one by one, to a growing initial seed assembly whenever the net glue strength of attachment exceeds some fixed temperature threshold. The aTAM has been shown to be capable of universal computation [15], and research leveraging this computational power has lead to efficient assembly of complex geometric shapes and patterns with a number of recent results in FOCS, SODA, and ICALP [1, 5–12, 14]. This universality also allows the model to serve directly as a model for computation in which an input bit string is encoded into an initial assembly. The process of self-assembly and the final produced terminal assembly represent the computation of a function on the given input. Given this framework, it is natural to ask how fast a given function can be computed in this model. Tile assembly systems can be designed to take advantage of massive parallelism when multiple tiles attach at distinct positions in parallel, opening the possibility for faster algorithms than what can be achieved in more traditional computational models. On the other hand, tile assembly algorithms must use up geometric space to perform computation, and must pay substantial time costs when communicating information between to physically distant bits. This creates a host of challenges unique to this physically motivated computational model that warrant careful study.

---

\*Department of Computer Science, University of Texas - Pan American, {abkeen, rtschweller, mjshe, zhongxingsi}@utpa.edu This author's research was supported in part by National Science Foundation Grant CCF-1117672.

Table 1: Summary of results.

	Time Complexity	Average Case
Addition(2D)	$\Theta(\sqrt{n})$ (Thm. 3.4, 6.1)	$O(\log n)$ (Thm.6.1)
Addition(3D)	$\Theta(\sqrt[3]{n})$ (Thm. 3.4, 8.1)	$O(\log n)$ (Thm.8.1)
Previous Best Addition(2D)	$O(n)$ [4]	-
Multiplication(d-D)	$\Omega(\sqrt[d]{n})$ (Thm. 3.6)	-

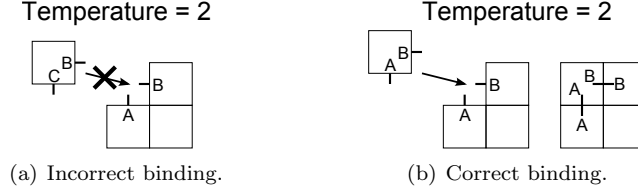


Figure 1: Cooperative tile binding in the aTAM.

In this paper we consider the time complexity of adding two  $n$ -bit numbers within the abstract tile assembly model. We show that this problem, along with multiplication, has a worst-case lower bound of  $\Omega(\sqrt{n})$  time in 2D and  $\Omega(\sqrt[3]{n})$  time in 3D. These lower bounds are derived by a reduction from a simple problem we term the *communication* problem in which two distant bits must compute the AND function between themselves. This general reduction technique can likely be applied to a number of problems and yields key insights into how one might design a sub-linear time solution to such problems. We in turn show how these lower bounds, in the case of 2D and 3D addition, are matched by corresponding worst case  $O(\sqrt{n})$  and  $O(\sqrt[3]{n})$  run time algorithms, respectively, which improves upon the previous best known result of  $O(n)$  [4]. We then consider the average case complexity of addition given two uniformly generated random  $n$ -bit numbers and construct a  $O(\log n)$  average case time algorithm that achieves simultaneous worst case run time  $O(\sqrt{n})$  in 2D. To the best of our knowledge this is the first tile assembly algorithm proposed for efficient average case adding. Our results are summarized in Table 1. In addition to our analytical results, tile self-assembly software simulations were conducted to visualize the diverse approaches to fast arithmetic presented in this paper, as well as to compare them to previous work. The adder tile constructions described in Sections 4, 5 and 6, and the previous best known algorithm from [4] were simulated using the two timing models described in Sections 2.3 and 2.3. These results can be seen in the graphs in Section 7.

## 2 Definitions

### 2.1 Basic Notation.

Let  $\mathbb{N}_n$  denote the set  $\{1, \dots, n\}$  and let  $\mathbb{Z}_n$  denote the set  $\{0, \dots, n-1\}$ . Consider two points  $p, q \in \mathbb{Z}^d$ ,  $p = (p_1, \dots, p_d)$ ,  $q = (q_1, \dots, q_d)$ . Define  $\Delta_{p,q} \triangleq \max_{1 \leq i \leq d} \{|p_i - q_i|\}$ .

### 2.2 Abstract Tile Assembly Model.

**Tiles.** Consider some alphabet of symbols  $\Pi$  called the *glue types*. A tile is a finite edge polygon (polyhedron in the case of a 3D generalization) with some finite subset of border points each assigned some glue type from  $\Pi$ . Further, each glue type  $g \in \Pi$  has some non-negative integer strength  $str(g)$ . For each tile  $t$  we also associate a finite string *label* (typically “0”, or “1”, or the empty label in this paper), denoted by  $label(t)$ , which allows the classification of tiles by their labels. In this paper we consider a special class of tiles that are unit squares (or unit cubes in 3D) of the same orientation with at most one glue type per face, with each glue being placed exactly in the center of the tile’s face. We denote the *location* of a tile to be the point at the center of the square or cube tile. In this paper we focus on tiles at integer locations.

**Assemblies.** An assembly is a finite set of tiles whose interiors do not overlap. Further, to simplify formalization in this paper, we further require the center of each tile in an assembly to be an integer coordinate (or integer triplet in 3D). If each tile in  $A$  is a translation of some tile in a set of tiles  $T$ , we say that  $A$  is an assembly over tile set  $T$ . For a given assembly  $\Upsilon$ , define the *bond graph*  $G_\Upsilon$  to be the weighted graph in which each element of  $\Upsilon$  is a vertex, and the weight of an edge between two tiles is the strength of the overlapping matching glue points between the two tiles. Note that only overlapping glues that are the same type contribute a non-zero weight, whereas overlapping, non-equal glues always contribute zero weight to the bond graph. The property that only equal glue types interact with each other is referred to as the *diagonal glue function* property and is perhaps more feasible than more general glue functions for experimental implementation. An assembly  $\Upsilon$  is said to be  $\tau$ -stable for an integer  $\tau$  if the min-cut of  $G_\Upsilon$  is at least  $\tau$ .

**Tile Attachment.** Given a tile  $t$ , an integer  $\tau$ , and a  $\tau$ -stable assembly  $A$ , we say that  $t$  may attach to  $A$  at temperature  $\tau$  to form  $A'$  if there exists a translation  $t'$  of  $t$  such that  $A' = A \cup \{t'\}$ , and  $A'$  is  $\tau$ -stable. For a tile set  $T$  we use notation  $A \rightarrow_{T,\tau} A'$  to denote that there exists some  $t \in T$  that may attach to  $A$  to form  $A'$  at temperature  $\tau$ . When  $T$  and  $\tau$  are implied, we simply say  $A \rightarrow A'$ . Further, we say that  $A \rightarrow^* A'$  if there exists a finite sequence of assemblies  $\langle A_1 \dots A_k \rangle$  such that  $A \rightarrow A_1 \rightarrow \dots \rightarrow A_k \rightarrow A'$ .

**Tile Systems.** A tile system  $\Gamma = (T, S, \tau)$  is an ordered triplet consisting of a set of tiles  $T$  referred to as the system's *tile set*, a  $\tau$ -stable assembly  $S$  referred to as the system's *seed* assembly, and a positive integer  $\tau$  referred to as the system's *temperature*. A tile system  $\Gamma = (T, S, \tau)$  has an associated set of *producible* assemblies,  $\text{PROD}_\Gamma$ , which define what assemblies can grow from the initial seed  $S$  by any sequence of temperature  $\tau$  tile attachments from  $T$ . Formally,  $S \in \text{PROD}_\Gamma$  as a base case producible assembly. Further, for any  $A \in \text{PROD}_\Gamma$ , if  $A \rightarrow_{T,\tau} A'$ , then  $A' \in \text{PROD}_\Gamma$ . That is, assembly  $S$  is producible, and for any producible assembly  $A$ , if  $A$  can grow into  $A'$ , then  $A'$  is also producible. We further define the set of *terminal* assemblies  $\text{TERM}_\Gamma$  to be the subset of  $\text{PROD}_\Gamma$  containing all producible assemblies that have no attachable tile from  $T$  at temperature  $\tau$ . Conceptually,  $\text{TERM}_\Gamma$  represents the final collection of output assemblies that are built from  $\Gamma$  given enough time for all assemblies to reach a final, terminal state. General tile systems may have terminal assembly sets containing 0, finite, or infinitely many distinct assemblies. Systems with exactly 1 terminal assembly are said to be *deterministic*. For a deterministic tile system  $\Gamma$ , we say  $\Gamma$  *uniquely assembles* assembly  $A$  if  $\text{TERM}_\Gamma = \{A\}$ . In this paper, we focus exclusively on deterministic systems. For recent consideration of non-determinism in tile self-assembly see [5–8, 12].

## 2.3 Problem Description.

We now formalize what we mean for a tile self-assembly system to compute a function. To do this we present the concept of a *tile assembly computer* (TAC) which consists of a tile set and temperature parameter, along with input and output *templates*. The input template serves as a seed structure with a sequence of *wildcard positions* for which tiles of label “0” and “1” may be placed to construct an initial seed assembly. An output template is a sequence of points denoting locations for which the TAC, when grown from a filled in template, will place tiles with “0” and “1” labels that denote the output bit string. A TAC then is said to compute a function  $f$  if for any seed assembly derived by plugging in a bitstring  $b$ , the terminal assembly of the system with tile set  $T$  and temperature  $\tau$  will be such that the value of  $f(b)$  is encoded in the sequence of tiles placed according to the locations of the output template. We now develop the formal definition of the TAC concept. We note that the formality in the input template is of substantial importance. Simpler definitions which map seeds to input bit strings, and terminal assemblies to output bitstrings, are problematic in that they allow for the possibility of encoding the computation of function  $f$  in the seed structure. Even something as innocuous sounding as allowing more than a single type of “0” or “1” tile as an input bit has the subtle issue of allowing pre-computing of  $f$ <sup>1</sup>.

<sup>1</sup>This subtle issue seems to exist with some previous formulations of tile assembly computation.

**Input Template.** Consider a tile set  $T$  containing exactly one tile  $t_0$  with label “0”, and one tile  $t_1$  with label “1”. An  $n$ -bit input template over tile set  $T$  is an ordered pair  $U = (R, B(i))$ , where  $R$  is an assembly over  $T - \{t_0, t_1\}$ ,  $B : \mathbb{N}_n \rightarrow \mathbb{Z}^2$ , and  $B(i)$  is not the position of any tile in  $R$  for any  $i$  from 1 to  $n$ . The sequence of  $n$  coordinates denoted by  $B$  conceptually denotes “wildcard” tile positions for which copies of  $t_0$  and  $t_1$  will be filled in for any instance of the template. For notation we define assembly  $U_b$  over  $T$ , for bit string  $b = b_1, \dots, b_n$ , to be the assembly consisting of assembly  $R$  unioned with a set of  $n$  tiles  $t^i$  for  $i$  from 1 to  $n$ , where  $t^i$  is equal a translation of tile  $t_{b(i)}$  to position  $B(i)$ . That is,  $U_b$  is the assembly  $R$  with each position  $B(i)$  tiled with either  $t_0$  or  $t_1$  according to the value of  $b_i$ .

**Output Template.** A  $k$ -bit output template is simply a sequence of  $k$  coordinates denoted by function  $C : \mathbb{N}_k \rightarrow \mathbb{Z}^2$ . For an output template  $V$ , an assembly  $A$  over  $T$  is said to represent binary string  $c = c_1, \dots, c_k$  over template  $V$  if the tile at position  $C(i)$  in  $A$  has label  $c_i$  for all  $i$  from 1 to  $k$ . Note that output template solutions are much looser than input templates in that there may be multiple tiles with labels “1” and “0”, and there are no restrictions on the assembly outside of the  $k$  specified wildcard positions. The strictness for the input template stems from the fact that the input must “look the same” in all ways except for the explicit input bit patterns. If this were not the case, it would likely be possible to encode the solution to the computational problem into the input template, resulting in a trivial solution.

**Function Computing Problem.** A *tile assembly computer* (TAC) is an ordered quadruple  $\mathfrak{S} = (T, U, V, \tau)$  where  $T$  is a tile set,  $U$  is an  $n$ -bit input template, and  $V$  is a  $k$ -bit output template. A TAC is said to compute function  $f : \mathbb{Z}_2^n \rightarrow \mathbb{Z}_2^k$  if for any  $b \in \mathbb{Z}_2^n$  and  $c \in \mathbb{Z}_2^k$  such that  $f(b) = c$ , then the tile system  $\Gamma_{\mathfrak{S},b} = (T, U_b, \tau)$  uniquely assembles an assembly  $A$  which represents  $c$  over template  $V$ . For a TAC  $\mathfrak{S}$  that computes the function  $f : \mathbb{Z}_2^{2n} \rightarrow \mathbb{Z}_2^{n+1}$  where  $f(r_1 \dots r_{2n}) = r_1 \dots r_n + r_{n+1} \dots r_{2n}$ , we say that  $\mathfrak{S}$  is an  $n$ -bit *adder* TAC with inputs  $a = r_1 \dots r_n$  and  $b = r_{n+1} \dots r_{2n}$ . An  $n$ -bit *multiplier* TAC is defined similarly.

**Run-time.** The run time model we use in this paper was first proposed in [3]. For a deterministic tile system  $\Gamma = (T, S, \tau)$  and assembly  $A \in \text{PROD}_\Gamma$ , the *1-step transition* set of assemblies for  $A$  is defined to be  $\text{STEP}_{\Gamma,A} = \{B \in \text{PROD}_\Gamma \mid A \rightarrow_{T,\tau} B\}$ . For a given  $A \in \text{PROD}_\Gamma$ , let  $\text{PARALLEL}_{\Gamma,A} = \bigcup_{B \in \text{STEP}_{\Gamma,A}} B$ , ie,  $\text{PARALLEL}_{\Gamma,A}$  is the result of attaching all singleton tiles that can attach directly to  $A$ . Note that since  $\Gamma$  is deterministic,  $\text{PARALLEL}_{\Gamma,A}$  is guaranteed to not contain overlapping tiles and is therefore an assembly. For an assembly  $A$ , we say  $A \rightrightarrows_\Gamma A'$  if  $A' = \text{PARALLEL}_{\Gamma,A}$ . We define the *parallel run-time* of a deterministic tile system  $\Gamma = (T, S, \tau)$  to be the non-negative integer  $k$  such that  $A_1 \rightrightarrows_\Gamma A_2 \rightrightarrows_\Gamma \dots \rightrightarrows_\Gamma A_k$  where  $A_1 = S$  and  $\{A_k\} = \text{TERM}_\Gamma$ . For any assemblies  $A$  and  $B$  in  $\text{PROD}_\Gamma$  such that  $A_1 \rightrightarrows_\Gamma A_2 \rightrightarrows_\Gamma \dots \rightrightarrows_\Gamma A_k$  with  $A = A_1$  and  $B = A_k$ , we say that  $A \rightrightarrows_\Gamma^k B$ . Alternately, we denote  $B$  with notation  $A \rightrightarrows_\Gamma^k B$ . For a TAC  $\mathfrak{S} = (T, U, V, \tau)$  that computes function  $f$ , the run time of  $\mathfrak{S}$  on input  $b$  is defined to be the parallel run-time of tile system  $\Gamma_{\mathfrak{S},b} = (T, U_b, \tau)$ . Worst case and average case run time are then defined in terms of the largest run time inducing  $b$  and the average run time for a uniformly generated random  $b$ .

**Probabilistic Run-time.** In addition to the simple parallel run time, we also introduce a probabilistic run time which is a discrete version of continuous time Markov process models originally proposed in [2] for the purposes of showing in simulation that our simple parallel model sufficiently captures the notion of run time when compared to a more realistic probabilistic model. Briefly, in a single time step the model attaches in parallel any number of tiles that may attach with the added restriction that each tile attaches with only probability  $\frac{1}{|T|}$ , modelling the concept of a run time in which a time step is normalized to the wait time for a single tile (not necessarily the correct tile) to bump into an attachment position. Analytical study of the relation between this type of run time and parallel run time is a direction for future work, although the two models should be within logarithmic factors of one another in expectation for constant size tile sets, which we utilize exclusively in this paper.

### 3 Lower Bound for Long Distance Communication

In this section we formulate a class of problems we term the *communication* problems in which the goal is to compute a simple AND function on a 2-bit input given that the input template separates the 2 input bits some specified distance  $\Delta$ . We formulate this problem for the purposes of providing lower bounds on the worst-case time complexity for this problem. We then reduce this problem to addition and multiplication problems in 2D and 3D to provide worst case lower bounds for addition and multiplication.

#### 3.1 High-Level Sketch of Lower Bound Proofs

To prove lower bounds for addition and multiplication in 2D and 3D, we do the following. First, we consider two identical tile systems with the exception of their respective seed assemblies which differ in exactly one tile location. We show in Lemma 3.1 that after  $\Delta$  time steps, all positions more than  $\Delta$  distance from the point of initial difference of the assemblies must be identical among the two systems. We then consider the *communication* problem in which we compute the AND function of two input bits under the assumption that the input template for the problem separates the two bits by distance  $\Delta$ . For such a problem, we know that the output position of the solution bit must be at least distance  $\frac{1}{2}\Delta$  from one of the two input bits. As the correct output for the AND function must be a function of both bits, Lemma 3.1 implies that at least  $\frac{1}{2}\Delta$  steps are required to guarantee a correct solution as argued in Theorem 3.2.

With the lower bound of  $\frac{1}{2}\Delta$  established for the communication problem, we move on to the problems of addition and multiplication of  $n$ -bit numbers. We show how the communication problem can be reduced to these problems, thereby yielding corresponding lower bounds. In particular, consider the addition problem in 2D. As the input template must contain positions for  $2n$  bits, in 2D it must be the case that some pair of bits are separated by at least  $\Omega(\sqrt{n})$  distance according to Lemma 3.3. Focussing on this pair of bit positions in the addition template, we can create a corresponding communication problem template with the same two positions as input. To guarantee the correct output, we hard code the remaining bit positions of the addition template such that the addition algorithm is guaranteed to place the AND of the desired bit pair in a specific position in the output template, thereby constituting a solution to the  $\Delta = \Omega(\sqrt{n})$  communication problem, which implies the addition solution cannot finish faster than  $\Omega(\sqrt{n})$  in the worst case. A similar reduction can be applied to multiplication. The precise reductions are detailed in Theorems 3.4 and 3.6.

#### 3.2 Communication Problem.

The  $\Delta$ -communication problem is the problem of computing the function  $f(b_1, b_2) = b_1 \wedge b_2$  for bits  $b_1$  and  $b_2$  in the 3D aTAM under the additional constraint that the input template for the solution  $U = (R, B(i))$  be such that  $\Delta = \max(|B(1)_x - B(2)_x|, |B(1)_y - B(2)_y|, |B(1)_z - B(2)_z|)$ .

We first establish a lemma which intuitively states that for any 2 seed assemblies that differ in only a single tile position, all points of distance greater than  $r$  from the point of difference will be identically tiled (or empty) after  $r$  time steps of parallelized tile attachments:

**Lemma 3.1.** Let  $S_{p,t}$  and  $S_{p,t'}$  denote two assemblies that are identical except for a single tile  $t$  versus  $t'$  at position  $p = (p_x, p_y, p_z)$  in each assembly. Further, let  $\Gamma = (T, S_{p,t}, \tau)$  and  $\Gamma' = (T, S_{p,t'}, \tau)$  be two deterministic tile assembly systems such that  $S_{p,t} \rightrightarrows_{\Gamma}^r R$  and  $S_{p,t'} \rightrightarrows_{\Gamma'}^r R'$  for non-negative integer  $r$ . Then for any point  $q = (q_x, q_y, q_z)$  such that  $r < \max(|p_x - q_x|, |p_y - q_y|, |p_z - q_z|)$ , it must be that  $R_q = R'_q$ , ie,  $R$  and  $R'$  contain the same tile at point  $q$ .

*Proof.* We show this by induction on  $r$ . As a base case of  $r = 0$ , we have that  $R = S_{p,t}$  and  $R' = S_{p,t'}$ , and therefore  $R$  and  $R'$  are identical at any point outside of point  $p$  by the definition of  $S_{p,t}$  and  $S_{p,t'}$ .

Inductively, assume that for some integer  $k$  we have that for all points  $w$  such that  $k < \Delta_{p,w} \triangleq \max(|p_x - w_x|, |p_y - w_y|, |p_z - w_z|)$ , we have that  $R_w^k = R'_w{}^k$ , where  $S_{p,t} \rightrightarrows_{\Gamma}^k R^k$ , and  $S_{p,t'} \rightrightarrows_{\Gamma'}^k R'^k$ . Now consider some point  $q$  such that  $k + 1 < \Delta_{p,q} \triangleq \max(|p_x - q_x|, |p_y - q_y|, |p_z - q_z|)$ , along with assemblies  $R^{k+1}$  and  $R'^{k+1}$  where  $S_{p,t} \rightrightarrows_{\Gamma}^{k+1} R^{k+1}$ , and  $S_{p,t'} \rightrightarrows_{\Gamma'}^{k+1} R'^{k+1}$ . Consider the direct neighbors (6 of them in 3D) of

point  $q$ . For each neighbor point  $c$ , we know that  $\Delta_{p,c} > k$ . Therefore, by inductive hypothesis,  $R_c^k = R_c'^k$  where  $S_{p,t} \rightrightarrows_{\Gamma}^k R^k$ , and  $S_{p,t'} \rightrightarrows_{\Gamma'}^k R'^k$ . Therefore, as attachment of a tile at a position is only dependent on the tiles in neighboring positions of the point, we know that tile  $R_q^{k+1}$  may attach to both  $R^k$  and  $R'^k$  at position  $q$ , implying that  $R_q^{k+1} = R_q'^{k+1}$  as  $\Gamma$  and  $\Gamma'$  are deterministic.  $\square$

**Theorem 3.2.** Any solution to the  $\Delta$ -communication problem has run time at least  $\frac{1}{2}\Delta$ .

*Proof.* Consider a TAC  $\mathfrak{S} = (T, U = (R, B(i)), V(i))$  that computes the  $\Delta$ -communication problem. First, note that  $B$  has domain of 1 and 2, and  $V$  has domain of just 1 (the input is 2 bits, the output is 1 bit). We now consider the value  $\Delta_V$  defined to be the largest distance between the output bit position of  $V$  from either of the two input bit positions in  $B$ : Let  $\Delta_V \triangleq \max(\Delta_{B(1),V(1)}, \Delta_{B(2),V(1)})$ . Without loss of generality, assume  $\Delta_V = \Delta_{B(1),V(1)}$ . Note that  $\Delta_V \geq \frac{1}{2}\Delta$ .

Now consider the computation of  $f(0,1) = 0$  versus the computation of  $f(1,1) = 1$  via our TAC  $\mathfrak{S}$ . Let  $A^0$  denote the terminal assembly of system  $\Gamma_0 = (T, U_{0,1}, \tau)$  and let  $A^1$  denote the terminal assembly of system  $\Gamma_1 = (T, U_{1,1}, \tau)$ . As  $\mathfrak{S}$  computes  $f$ , we know that  $A_{V(1)}^0 \neq A_{V(1)}^1$ . Further, from Lemma 3.1, we know that for any  $r < \Delta_V$ , we have that  $W_{V(1)}^0 = W_{V(1)}^1$  for any  $W^0$  and  $W^1$  such that  $U_{0,1} \rightrightarrows_{\Gamma_0}^r W^0$  and  $U_{1,1} \rightrightarrows_{\Gamma_1}^r W^1$ . Let  $d_{\mathfrak{S}}$  denote the run time of  $\mathfrak{S}$ . Then we know that  $U_{0,1} \rightrightarrows_{\Gamma_0}^{d_{\mathfrak{S}}} A^0$ , and  $U_{1,1} \rightrightarrows_{\Gamma_1}^{d_{\mathfrak{S}}} A^1$  by the definition of run time. If  $d_{\mathfrak{S}} < \Delta_V$ , then Lemma 3.1 implies that  $A_{V(1)}^0 = A_{V(1)}^1$ , which contradicts the fact that  $\mathfrak{S}$  compute  $f$ . Therefore, the run time  $d_{\mathfrak{S}}$  is at least  $\Delta_V \geq \frac{1}{2}\Delta$ .  $\square$

### 3.3 $\Omega(\sqrt[d]{n})$ Lower Bounds for Addition and Multiplication.

We now show how to reduce instances of the communication problem to the arithmetic problems of addition and multiplication in 2D and 3D to obtain lower bounds of  $\Omega(\sqrt[n]{n})$  and  $\Omega(\sqrt[3]{n})$  respectively.

We first show the following Lemma which lower bounds the distance of the farthest pair of points in a set of  $n$  points. We believe this Lemma is likely well known, or is at least the corollary of an established result. We include it's proof for completeness.

**Lemma 3.3.** For positive integers  $n$  and  $d$ , consider  $A \subset \mathbb{Z}^d$  and  $B \subset \mathbb{Z}^d$  such that  $A \cap B = \emptyset$  and  $|A| = |B| = n$ . There must exist points  $p \in A$  and  $q \in B$  such that  $\Delta_{p,q} \geq \lceil \frac{1}{2} \lceil \sqrt[d]{2n} \rceil \rceil - 1$ .

*Proof.* To see this, consider a bounding box of all  $2n$  points. If all  $d$  dimensions of the bounding box were of length strictly less than  $\sqrt[d]{2n}$ , then the box could not contain all  $2n$  points. Therefore, at least one dimension is of length at least  $\lceil \sqrt[d]{2n} \rceil$ , implying that there are two points of distance at least  $\lceil \sqrt[d]{2n} \rceil - 1$  along that particular axis. If these two points are in  $A$  and  $B$  respectively, then the claim follows. If not, say both are from set  $A$ , then there must be a point in  $B$  that is at least  $\lceil \frac{1}{2} \lceil \sqrt[d]{2n} \rceil \rceil - 1$  from one of these two points in  $A$ , implying the claim.  $\square$

**Theorem 3.4.** Any  $n$ -bit adder TAC that has a dimension  $d$  input template for  $d = 1$ ,  $d = 2$ , or  $d = 3$ , has a worst case run time of  $\Omega(\sqrt[d]{n})$ .

*Proof.* To show the lower bound, we will reduce the  $\Delta$ -communication problem for some  $\Delta = \Omega(\sqrt[d]{n})$  to the  $n$ -bit adder problem with a  $d$ -dimension template. Consider some  $n$ -bit adder TAC  $\mathfrak{S} = (T, U = (F, W), V, \tau)$  such that  $U$  is a  $d$ -dimension template. The  $2n$  sequence of wildcard positions  $W$  of this TAC must be contained in  $d$ -dimensional space by the definition of a  $d$ -dimension template, and therefore by Lemma 3.3 there must exist points  $W(i)$  for  $i \leq n$ , and  $W(n+j)$  for  $j \leq n$ , such that  $\Delta_{W(i),W(n+j)} \geq \lceil \frac{1}{2} \lceil \sqrt[d]{2n} \rceil \rceil - 1 = \Omega(\sqrt[d]{n})$ . Now consider two  $n$ -bit inputs  $a = a_n \dots a_1$  and  $b = b_n \dots b_1$  to the adder TAC  $\mathfrak{S}$  such that:  $a_k = 0$  for any  $k > i$  and any  $k < j$ , and  $a_k = 1$  for any  $k$  such that  $j \leq k < i$ . Further, let  $b_k = 0$  for all  $k \neq j$ . The remaining bits  $a_i$  and  $a_j$  are unassigned variables of value either 0 or 1. Note that the  $i+1$  bit of  $a+b$  is 1 if and only if  $a_i$  and  $b_j$  are both value 1. This setup constitutes our reduction of the  $\Delta$ -communication problem to the addition problem as the adder TAC template with the specified bits hardcoded in constitutes a template for the  $\Delta$ -communication problem that produces the AND of the input bit pair. We now specify explicitly how to generate a communication TAC from a given adder TAC.

For given  $n$ -bit adder TAC  $\mathfrak{S} = (T, U = (F, W), V, \tau)$  with dimension  $d$  input template, we derive a  $\Delta$ -communication TAC  $\rho = (T, U^2 = (F^2, W^2), V^2, \tau)$  as follows. First, let  $W^2(1) = W(i)$ , and  $W^2(2) = W(n+j)$ . Note that as  $\Delta_{W(i), W(n+j)} = \Omega(\sqrt[d]{n})$ ,  $W^2$  satisfies the requirements for a  $\Delta$ -communication input template for some  $\Delta = \Omega(\sqrt[d]{n})$ . Derive the frame of the template  $F^2$  from  $F$  by adding tiles to  $F$  as follows: For any positive integer  $k > i$ , or  $k < j$ , or  $k > n$  but not  $k = n+j$ , add a translation of  $t_0$  (with label “0”) translated to position  $W(k)$ . Additionally, for any  $k$  such that  $j \leq k < i$ , add a translation of  $t_1$  (with label “1”) at translation  $W(k)$ .

Now consider the  $\Delta$ -communication TAC  $\rho = (T, U^2 = (F^2, W^2), V^2, \tau)$  for some  $\Delta = \Omega(\sqrt[d]{n})$ . As assembly  $U_{a_i, b_j}^2 = U_{a_1 \dots a_n, b_1 \dots b_n}$ , we know that the worst case run time of  $\rho$  is at most that of the worst case run time of  $\mathfrak{S}$ . Therefore, by Theorem 3.2, we have that  $\mathfrak{S}$  has a run time of at least  $\Omega(\sqrt[d]{n})$ .  $\square$

As the bound of dimension  $d$  on the input template of a TAC alone lower bounds the run time of the TAC, we get the following corollary.

**Corollary 3.5.** Any  $d$ -dimension  $n$ -bit adder TAC has worst case run-time  $\Omega(\sqrt[d]{n})$ .

We now provide a lower bound for multiplication.

**Theorem 3.6.** Any  $n$ -bit multiplier TAC that has a dimension  $d$  input template for  $d = 1$ ,  $d = 2$ , or  $d = 3$ , has a worst case run time of  $\Omega(\sqrt[d]{n})$ .

*Proof.* Consider some  $n$ -bit multiplier TAC  $\mathfrak{S} = (T, U = (F, W), V, \tau)$  with  $d$ -dimension input template. By Lemma 3.3, some  $W(i)$  and  $W(n+j)$  must have distance at least  $\Delta \geq \lceil \frac{1}{2} \lceil \sqrt[d]{2n} \rceil \rceil - 1$ . Now consider input strings  $a = a_n \dots a_1$  and  $b = b_n \dots b_1$  to  $\mathfrak{S}$  such that  $a_i$  and  $b_j$  are of variable value, and all other  $a_k$  and  $b_k$  have value 0. For such input strings, the  $i+j$  bit of the product  $ab$  has value 1 if and only if  $a_i = b_j = 1$ . Thus, we can convert the  $n$ -bit multiplier system  $\mathfrak{S}$  into a  $\Delta$ -communication TAC with the same worst case run time in the same fashion as for Theorem 3.4, yielding a  $\Omega(\sqrt[d]{n})$  lower bound for the worst case run time of  $\mathfrak{S}$ .  $\square$

As with addition, the lower bound implied by the limited dimension of the input template alone yields the general lower bound for  $d$  dimensional multiplication TACS.

**Corollary 3.7.** Any  $d$ -dimension  $n$ -bit multiplier TAC has worst case run-time  $\Omega(\sqrt[d]{n})$ .

## 4 Addition In Average Case Logarithmic Time

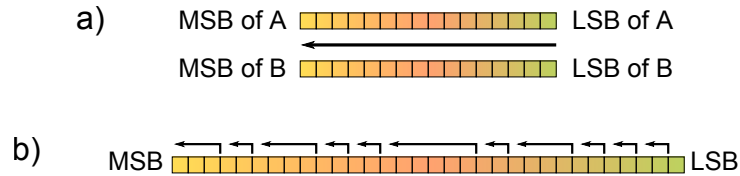


Figure 2: Arrows represent carry origination and propagation direction. a) This schematic represents the previously described  $O(n)$  worst case addition for addends  $A$  and  $B$  [4]. The least significant and most significant bits of  $A$  and  $B$  are denoted by LSB and MSB, respectively. b) The average case  $O(\log n)$  construction described in this paper is shown here. Addends  $A$  and  $B$  populate the linear assembly with bit  $A_i$  immediately adjacent to  $B_i$ . Carry propagation is done in parallel along the length of the assembly.

We construct an adder TAC that resembles an electronic carry-skip adder in that the carry-out bit for addend pairs where each addend in the pair has the same bit value is generated in a constant number of steps and immediately propagated. When each addend in a pair of addends does not have the same bit value, a carry-out cannot be deduced until the value of the carry-in to the pair of addends is known. When such

addends combinations occur in a contiguous sequence, the carry must ripple through the sequence from right-to-left, one step at a time as each position is evaluated. Within these worst-case sequences, our construction resembles an electronic ripple-carry adder. We show that using this approach it is possible to construct an  $n$ -bit adder TAC that can perform addition with an average runtime of  $O(\log n)$  and a worst-case runtime of  $O(n)$ .

**Lemma 4.1.** Consider a non-negative integer  $N$  generated uniformly at random from the set  $\{0, 1, \dots, 2^n - 1\}$ . The expected length of the longest substring of contiguous ones in the binary expansion of  $N$  is  $O(\log n)$ .

For a proof of Lemma 4.1 please see Schilling [13].

**Theorem 4.2.** For any positive integer  $n$ , there exists an  $n$ -bit adder TAC (tile assembly computer) that has worst case run time  $O(n)$  and an average case run time of  $O(\log n)$ .

The proof of Theorem 4.2 follows from the construction of the adder in Sections 4.1, 4.2, and 4.3.

## 4.1 Construction

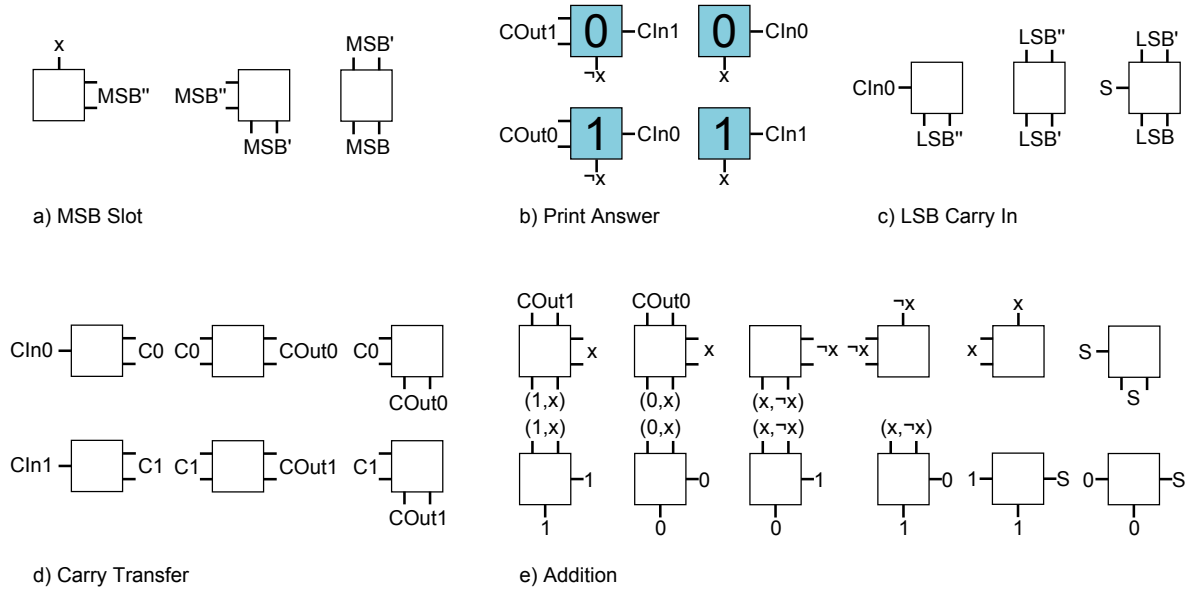


Figure 3: This is the complete set of tiles necessary to implement average case  $O(\log n)$  and worst case  $O(n)$  addition.

We summarize the mechanism of addition presented here in a short example. The complete tile set may be found in Figure 3.

**Input Template.** The input template, or seed, for the construction of an adder with a  $O(\log n)$  average case is shown in Figure 4. This input template is composed of  $n$  blocks, each containing three tiles. Within a block, the easternmost tile is the  $S$  labeled tile followed by two tiles representing  $A_k$  and  $B_k$ , the  $k$ th bits of  $A$  and  $B$  respectively. Of these  $n$  blocks, the easternmost and westernmost blocks of the template assembly are unique. Instead of an  $S$  tile, the block furthest east has an  $LSB$ -labeled tile which accompanies the tiles representing the least significant bits of  $A$  and  $B$ ,  $A_0$  and  $B_0$ . The westernmost block of the template assembly contains a block labeled  $MSB$  instead of the  $S$  block and accompanies the most significant bits of  $A$  and  $B$ ,  $A_{n-1}$  and  $B_{n-1}$ .



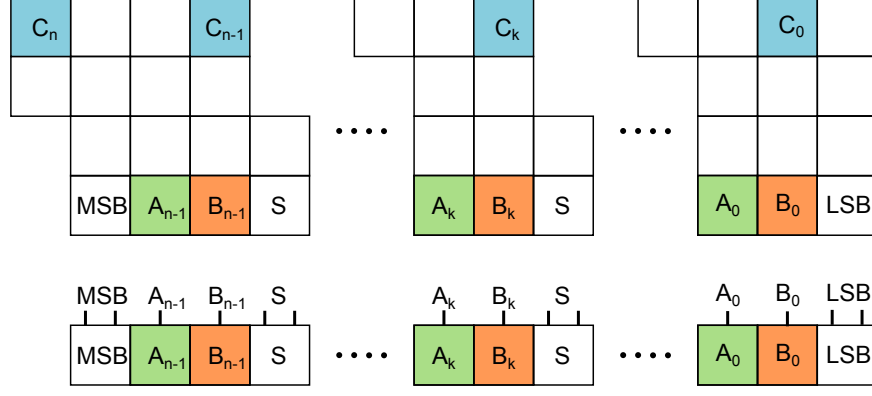


Figure 4: Top: Output template displaying addition result  $C$  for  $O(\log n)$  average case addition construction. Bottom: Input template composed of  $n$  blocks of three tiles each, representing  $n$ -bit addends  $A$  and  $B$ .

**Computing Carry Out Bits.** For clarity, we demonstrate the mechanism of this adder TAC through an example by selecting two 4-bit binary numbers  $A$  and  $B$  such that the addends  $A_i$  and  $B_i$  encompass every possible addend combination. The input template for such an addition is shown in Figure 5a where orange tiles represent bits of  $A$  and green tiles represent bits of  $B$ . Each block begins the computation in parallel at each  $S$  tile. After six parallel steps (Figure 5b-g), all carry out bits, represented by glues  $C0$  and  $C1$ , are determined for addend-pairs where both  $A_i$  and  $B_i$  are either both 0 or both 1. For addend pairs  $A_i$  and  $B_i$  where one addend is 0 and one addend is 1, the carry out bit cannot be deduced until a carry out bit has been produced by the previous addend pair,  $A_{i-1}$  and  $B_{i-1}$ . By step seven, a carry bit has been presented to all addend pairs that are flanked on the east by an addend pair comprised of either both 0s or both 1s, or that are flanked on the east by the  $LSB$  start tile, since the carry in to this site is always 0 (Figure 5h). For those addend pairs flanked on the east by a contiguous sequence of size  $j$  pairs consisting of one 1 and one 0,  $2j$  parallel attachment steps must occur before a carry bit is presented to the pair.

**Computing the Sum.** Once a carry out bit has been computed and carried into an addend pair  $A_i$  and  $B_i$ , two parallel tile addition steps are required to compute the sum of the addend pair (Figure 5g-j).

## 4.2 Time Complexity.

**$O(n)$  - worst case.** We first show that this construction has a  $O(n)$  worst case run-time under the timing model presented in Section 2.3 *Run-time*. Consider a binary sequence  $T$  of length  $2n$  representing two  $n$ -bit binary numbers  $A$  and  $B$ .  $A_0$  and  $B_0$  represent the least significant bits of  $A$  and  $B$ , respectively, and  $A_n$  and  $B_n$  represent the most significant bits of  $A$  and  $B$ , respectively. The formatting of  $T$  is such that  $T_i = B_{i/2}$  if  $i$  is even, and  $T_i = A_{(i-1)/2}$  if  $i$  is odd. Sequence  $T$  is shown in Figure 7a.

$T$  contains  $n$  addend-pairs,  $(A_i, B_i)$ , which are ordered pairs consisting of the  $i$ th bit of  $A$  and the  $i$ th bit of  $B$ . The four possible values for each addend-pair are shown in Figure 7b, along with the carry bits they produce upon addition. In  $T$ , there exist sequences of various sizes up to  $k$  addend-pairs such that every addend-pair in the interval from  $(A_i, B_i)$  to  $(A_{i+j-1}, B_{i+j-1})$  of a size  $j$  addend-pair sequence is  $(1, 0)$  or  $(0, 1)$ . In the adder TAC outlined in Section 4 and Appendix 4.1, the value of the carry bit produced upon addition of  $A_i + B_i$  for every  $(0, 0)$  and  $(1, 1)$  addend-pair is known and made available to the next addend-pair  $(A_{i+1}, B_{i+1})$  after seven parallel tile addition steps, including the carry bit into  $(A_0, B_0)$ . Therefore, after a constant number of parallel tile addition steps, the first addend-pair  $(A_i, B_i)$  in any  $j$  addend-pair-length sequence of  $(1, 0)$  or  $(0, 1)$  addend-pairs will be presented with the carry bit from the previous addend-pair  $(A_{i-1}, B_{i-1})$ . After  $2j$  subsequent parallel tile addition steps, the carry bit from the last addend-pair  $(A_{i+j-1}, B_{i+j-1})$  of the  $j$ -size sequence of consecutive  $(0, 1)$  and  $(1, 0)$  addend-pairs is presented to  $(A_{i+j}, B_{i+j})$ . Once an addend-pair has received a carry-in bit, the final sum is computed in

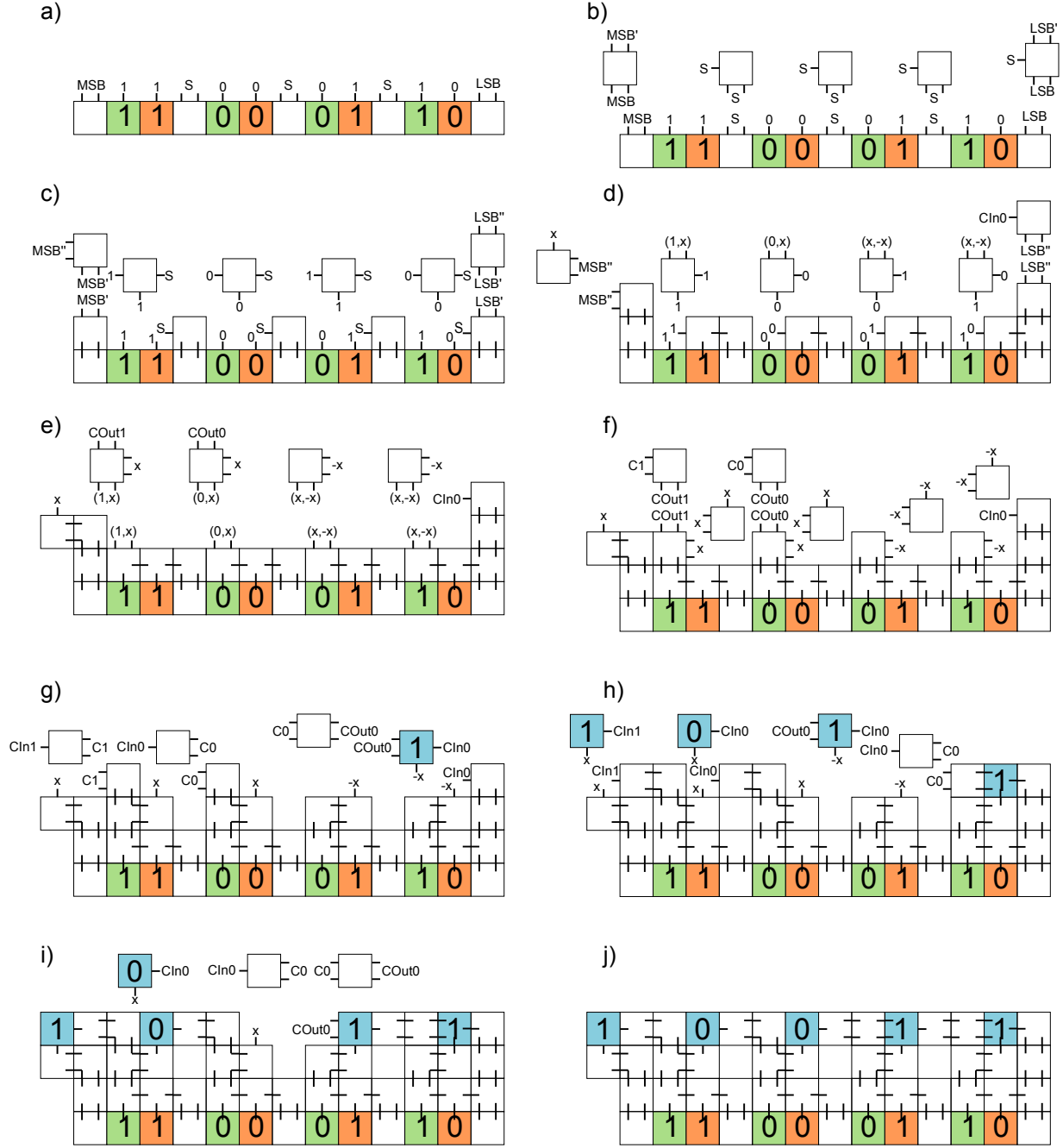
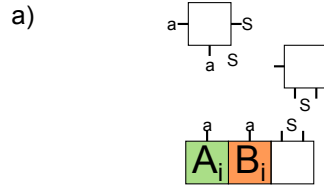
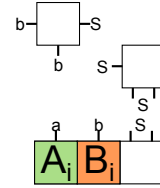


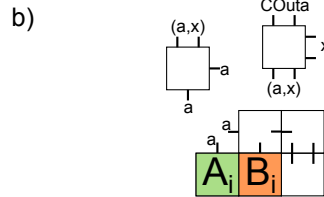
Figure 5: Example tile assembly system to compute the sum of 1001 and 1010 using the adder TAC presented in Section 4. a) The template with the two 4-bit binary numbers to be summed. b) The first step is the parallel binding of tiles to the  $S$ ,  $LSB$ , and  $MSB$  tiles. c) Tiles bind cooperatively to west-face  $S$  glues and glues representing bits of  $B$ . The purpose of this step is to propagate bit  $B_i$  closer to  $A_i$  so that in d) a tile may bind cooperatively, processing information from both  $A_i$  and  $B_i$ . e) Note that addend-pairs consisting of either both 1s or both 0s have a tile with a north face glue consisting of either  $(1,x)$  or  $(0,x)$  bound to the  $A_i$  tile. This glue represents a carry out of either 1 or 0 from the addend-pair. In (e-g) the carry outs are propagated westward via tile additions for those addend-pairs where the carry out can be determined. Otherwise, spacer tiles bind. h) Tiles representing bits of  $C$  (the sum of  $A$  and  $B$ ) begin to bind where a carry in is known. i-j) As carry bits propagate through sequences of  $(0,1)$  or  $(1,0)$  addend pairs, the final sum is computed.



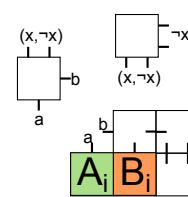
$A_i$  and  $B_i$  are both 0s or both 1s.



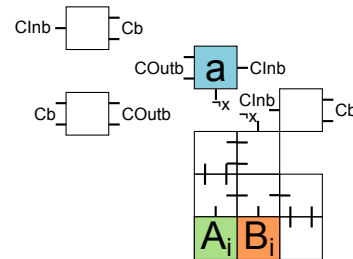
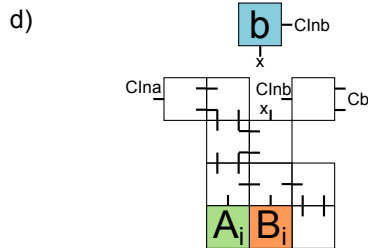
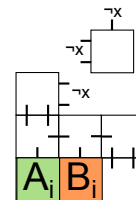
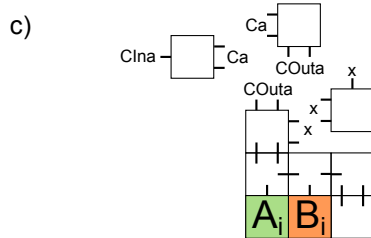
$A_i$  and  $B_i$  are different;  $a=0, b=1$  or  $a=1, b=0$ .



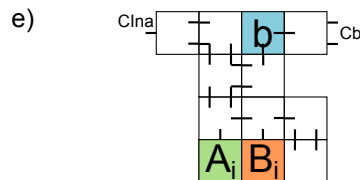
The  $(a,x)$  means that the value of the carry generated for next block is  $a$  and the result in this block will be the value of the carry-in,  $x$ , from previous block.



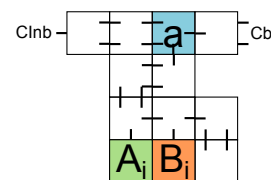
The  $(x, \neg x)$  means that the value of the carry which will be propagated to the next block is  $x$ , which is same as the carry-in from the previous block. Also, the result in this block will be NOT  $x$ .



Here we see that if the input values are equal, the carry that was just generated will be sent to the next block immediately. However, if the input values are not equal, the carry for next block will be propagated after the carry-in is accepted.



$A_i=B_i=a$ ; carry= $b$ ;  $A_i+B_i+\text{carry}=ab$



$A_i=\neg B_i$ ;  $a=\neg b$ ; carry= $b$ ;  $A_i+B_i+\text{carry}=ba$

Figure 6: The set of figures on the left side show how a carry out may be generated before the *addend-pair* has recieved a carry in. The set of figures on the right side show how a carry out can be dependent upon a carry in having been recieved by the *addend-pair*.

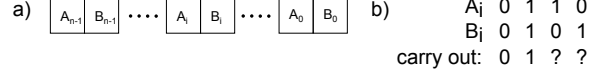


Figure 7: a) Binary sequence  $T$  populated by  $n$ -bit binary numbers  $A$  and  $B$ . b) Four possible  $(A_i, B_i)$  addend-pair combinations.

one tile addition step. If  $k$  is the longest contiguous sequence of  $(0, 1)$  and  $(1, 0)$  addend-pairs, then  $2k + 8$  parallel tile addition steps are required to compute the sum  $C$  of  $A$  and  $B$  (Figure 8). Therefore, the time complexity is  $O(k)$ . Since the growth is bounded upwards by the longest contiguous sequence  $k$  of  $(0, 1)$  and

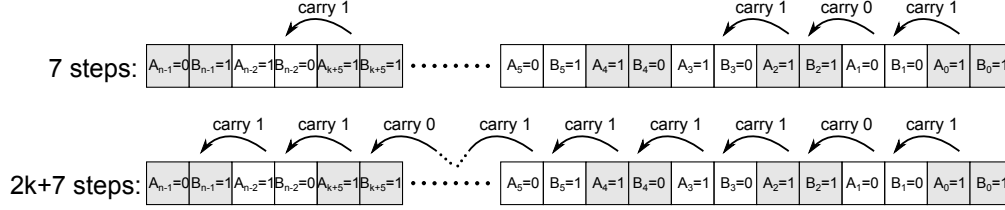


Figure 8:

$(1, 0)$  addend-pairs, then the worst-case scenario occurs when  $k = n$ . Thus, the worst-case run time is  $O(n)$ .

**$O(\log n)$  - average case.** We now show that the average case run-time is  $O(\log n)$  under the timing model presented in Section 2.3. In two  $n$ -bit randomly generated binary numbers,  $A$  and  $B$ , the probability of one of the  $(1, 0)$   $(0, 1)$  addend-pair cases occurring at  $(A_i, B_i)$  is  $1/2$ . The sequence of  $(A_i, B_i)$  bit pairs can thus be thought of as a Bernoulli process in which the likelihood of occurrence of a  $(0, 1)$  or  $(1, 0)$  addend-pair is equal to the occurrence of a  $(1, 1)$  or  $(0, 0)$  addend-pair. As shown above, the runtime is bounded by  $k$ , the longest contiguous sequence of  $(0, 1)$  or  $(1, 0)$  addend-pairs, which might be thought of as the longest sequence of heads in  $n$  independent fair coin tosses. Using Lemma 4.1, the expected longest run of heads in  $n$  coin tosses is  $O(\log n)$  [13]. Therefore, the average case time complexity of the tile addition algorithm described above is  $O(\log n)$ .

### 4.3 Correctness.

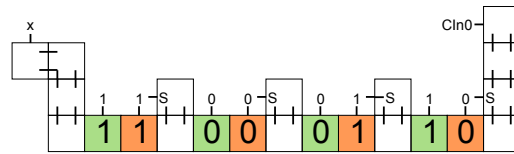


Figure 9: The scaffolding needed for the computation.

The analysis for the average case  $O(\log n)$  worst case  $O(n)$  adder TAC will begin from the point of the construction after which all scaffolding is in place (Figure 9).

The first step for any addend-pair in the seed is to perform its addition. This addition step will output either  $(1, x)$ ,  $(0, x)$ , or  $(x, \neg x)$  according to the following formulas:

$$\begin{aligned} 1 + 1 &= (1, x) \\ 0 + 0 &= (0, x) \\ 1 + 0 &= (x, \neg x) \end{aligned}$$

$$0 + 1 = (x, \neg x)$$

The first value in the output represents the carry-out of the addend-pair and the second value represents the value of the addition. In both cases,  $x$  refers to the unknown carry-in which will come from the previous addend-pair's carry out.

Given any addend-pair, we can then determine whether or not we have enough information to generate a carry by looking at the first position of the generated output pair in the first step. The two cases that can generate a carry,  $(1, x)$  and  $(0, x)$ , do just that and immediately propagate a 1 or 0 respectively. The other case,  $(x, \neg x)$ , simply waits for a carry to come in. No addend-pair can calculate its value until it receives a carry from the previous addend-pair. Once an addend-pair receives a carry-in, it can replace any  $x$  or  $\neg x$  with the proper value and it can “print” the correct value, i.e. the solution at that particular position. Also, if  $k_0$  was  $x$  it can now propagate its carry to the next addend-pair.

## 5 Optimal $O(\sqrt{n})$ Addition

We show how to construct an adder TAC that achieves a run time of  $O(\sqrt{n})$ , which matches the lower bound proved in Theorem 3.4. This adder TAC closely resembles an electronic carry-select adder in that the addends are divided into sections of size  $\sqrt{n}$  and the sum of the addends comprising each is computed for both possible carry-in values. The correct result for the subsection is then selected after a carry-out has been propagated from the previous subsection. Within each subsection, the addition scheme resembles a ripple-carry adder. This construction works well with massive parallelism and allows us to construct an optimal  $O(\sqrt{n})$  adder TAC in two dimensions.

**Theorem 5.1.** There exists a 2D  $n$ -bit adder TAC with a worst case run-time of  $O(\sqrt{n})$ .

The proof of Theorem 5.1 follows from the construction of the tile assembly adder in Sections 5.1, 5.2, and 5.3.

### 5.1 Construction.

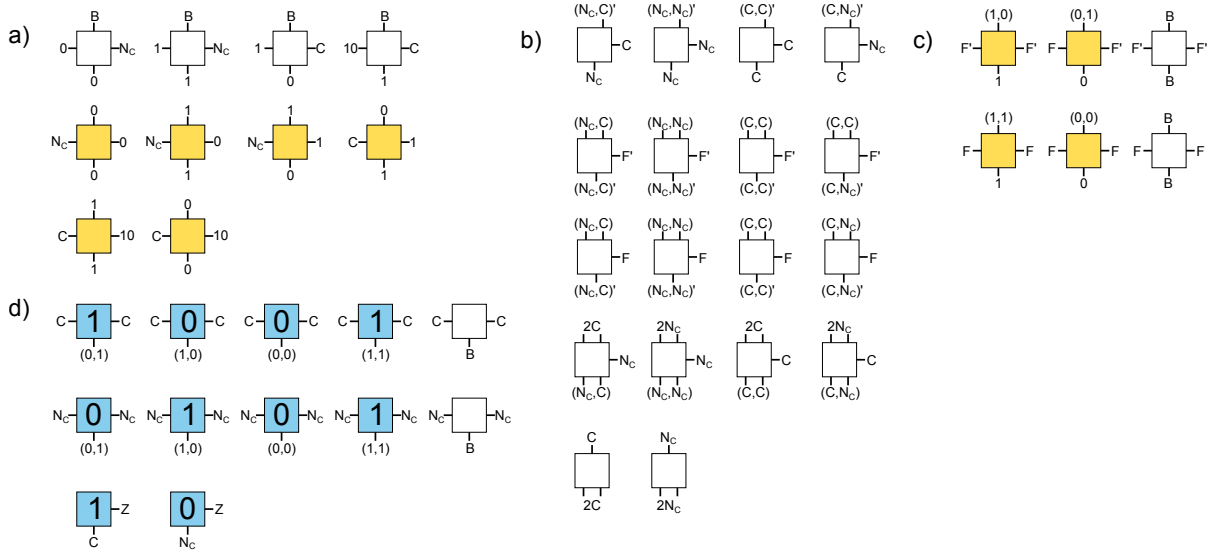


Figure 10: The complete tile set. a) Tiles involved in bit addition. b) Carry propagation tiles. c) Tiles involved in incrementation. d) Tiles that print the answer.

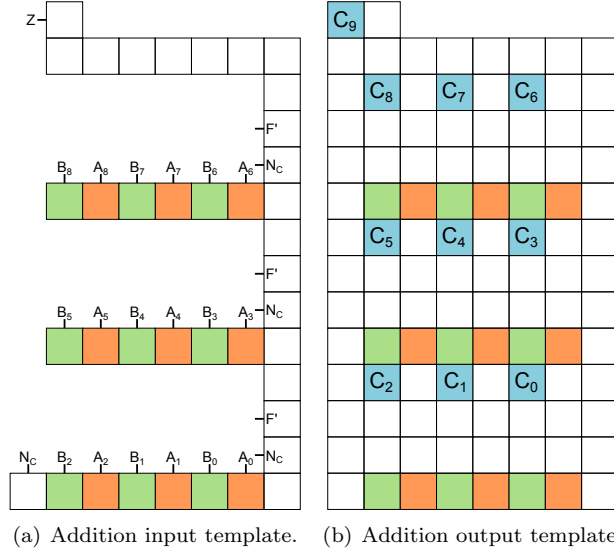


Figure 11: These are example I/O templates for the worst case  $O(\sqrt[2]{n})$  time addition introduced in Section 5.

**Input/Output Template.** Figure 11(a) and Figure 11(b) are examples of I/O templates for a 9-bit adder TAC. The inputs to the addition problem in this instance are two 9-bit binary numbers  $A$  and  $B$  with the least significant bit of  $A$  and  $B$  represented by  $A_0$  and  $B_0$ , respectively. The north facing glues in the form of  $A_i$  or  $B_i$  in the input template must either be a 1 or a 0 depending on the value of the bit in  $A_i$  or  $B_i$ . The placement for these tiles is shown in Figure 11(a) while a specific example of a possible input template is shown in Figure 12a. The sum of  $A + B$ ,  $C$ , is a ten bit binary number where  $C_0$  represents the least significant bit. The placement for the tiles representing the result of the addition is shown in Figure 11(b) while a specific example of an output is shown in Figure 14j.

To construct an  $n$ -bit adder in the case that  $n$  is a perfect square, split the two  $n$ -bit numbers into  $\sqrt{n}$  sections each with  $\sqrt{n}$  bits. Place the bits for each of these two numbers as per the previous paragraph, except with  $\sqrt{n}$  bits per row, making sure to alternate between  $A$  and  $B$  bits. There will be the same amount of space between each row as seen in the example template 11(a). All  $Z$ ,  $N_C$ , and  $F'$ , must be placed in the same relative locations. The solution,  $C$ , will be in the output template s.t.  $C_i$  will be three tile positions above  $B_i$  and a total of size  $n + 1$ .

Below, we use the adder tile set to add two nine-bit numbers:  $A = 100110101$  and  $B = 110101100$  to demonstrate the three stages in which the adder tile system performs addition.

**Step One: Addition.** With the inclusion of the seed assembly (Figure 12a) to the tile set (Figure 10), the first subset of tiles able to bind are the addition tiles shown in Figure 10a. These tiles sum each pair of bits from  $A$  and  $B$  (for example,  $A_0 + B_0$ ) (Figure 12b). Tiles shown in yellow are actively involved in adding and output the sum of each bit pair on the north face glue label. Yellow tiles also output a carry to the next more significant bit pair, if one is needed, as a west face glue label. Spacer tiles (white) output a  $B$  glue on the north face and serve only to propagate carry information from one set of  $A$  and  $B$  bits to the next. Each row computes this addition step independently and outputs a carry or no-carry west face glue on the westernmost tile of each row (Figure 12c). In a later step, this carry or no-carry information will be propagated northwards from the southernmost row in order to determine the sum. Note that immediately after the first addition tile is added to a row of the seed assembly, a second layer may form by the attachment of tiles from the increment tile set (Figure 10)c. While these two layers may form nearly concurrently, we separate them in this example for clarity and instead address the formation of the second layer of tiles in

Step Two: Increment below.

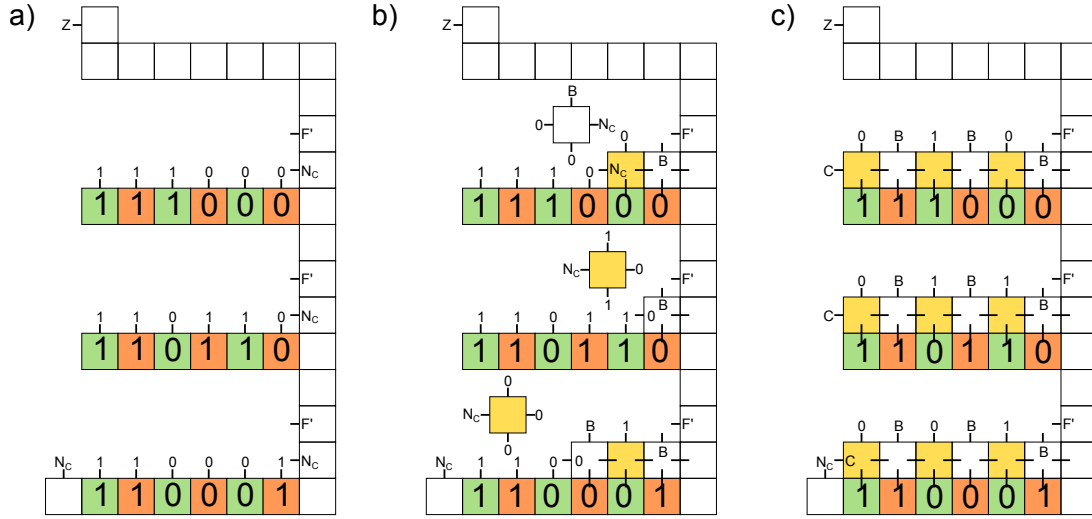


Figure 12: Step 1: Addition.

**Step Two: Increment.** As the addition tiles bind to the seed, tiles from the incrementation tile set (Figure 10c) may also begin to cooperatively attach. For clarity, we show their attachment following the completion of the addition layer. The purpose of the incrementation tiles is to determine the sum for each  $A$  and  $B$  bit pair in the event of a no-carry from the row below and in the event of a carry from the row below (Figure 13). The two possibilities for each bit pair are presented as north facing glues on yellow increment tiles. These north face glues are of the form  $(x, y)$  where  $x$  represents the value of the sum in the event of no-carry from the row below while  $y$  represents the value of the sum in the event of a carry from the row below. White incrementation tiles are used as spacers, with the sole purpose of passing along carry or no-carry information via their east/west face glues  $F'$ , which represents a no-carry, and  $F$ , which represents a carry.

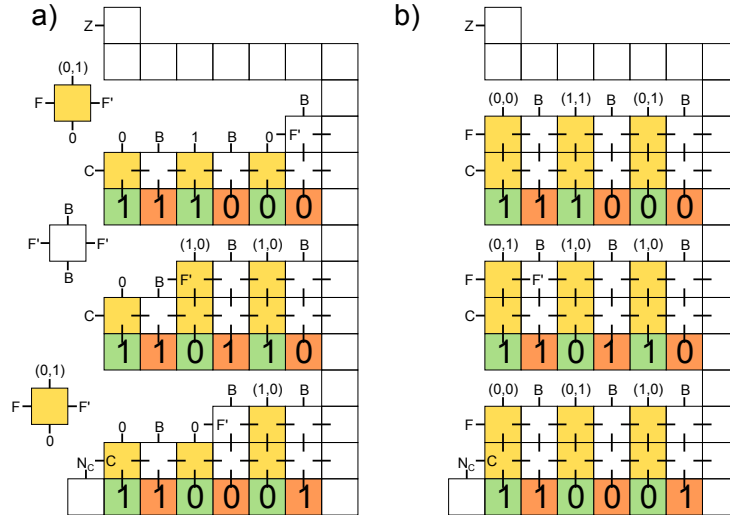


Figure 13: Step 2: Increment.

**Step Three: Carry Propagation and Output.** The final step of the addition mechanism presented here propagates carry or no-carry information northwards from the southernmost row of the assembly using tiles from the tile set in Figure 10b and then outputs the answer using the tile set in Figure 10d. Following completion of the incrementation layers, tiles may begin to grow up the west side of the assembly as shown in Figure 14a. When the tiles grow to a height such that the empty space above the increment row is presented with a carry or no-carry as in Figure 14b, the output tiles may begin to attach from west to east to print the answer (Figure 14c). As the carry propagation column grows northwards and presents carry or no carry information to each empty space above each increment layer, the sum may be printed for each row Figures 14d-e. When the carry propagation column reaches the top of the assembly, the most significant bit of the sum may be determined and the calculation is complete (Figure 14f).

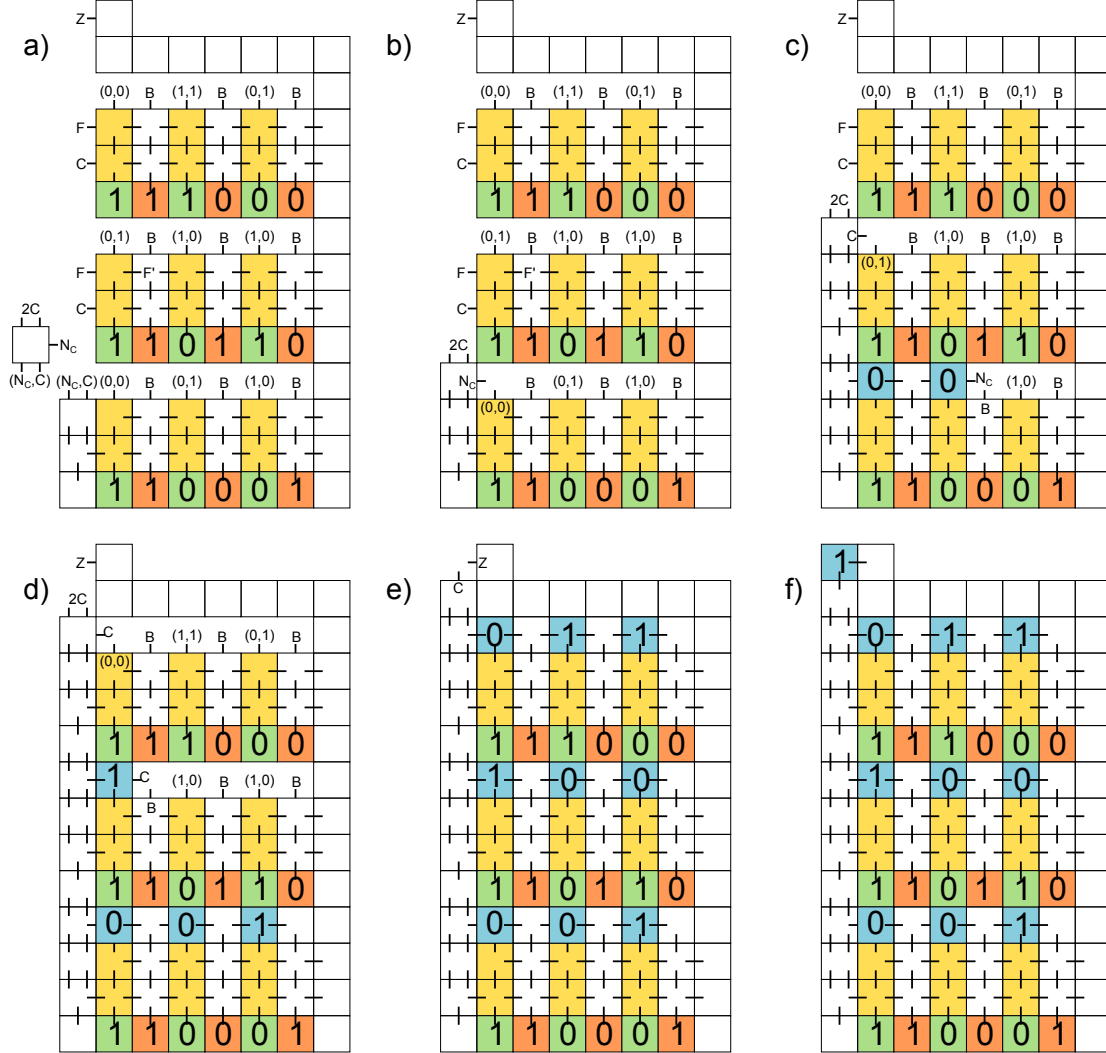


Figure 14: Step 3: Carry Propagation and Output.

## 5.2 Time Complexity.

Using the runtime model presented in Section 2.3 *Run-time* we will show that the addition algorithm presented in this section has a worst case runtime of  $O(\sqrt{n})$ . In order to ease the analysis we will assume that



each logical step of the algorithm happens in a synchronized fashion even though parts of the algorithm are running concurrently.

The first step of the algorithm is the addition of two  $O(\sqrt{n})$  numbers co-located on the same row. This first step occurs by way of a linear growth starting from the leftmost bit all the way to the rightmost bit of the row. The growth of a line one tile at a time has a runtime on the order of the length of the line. In the case of our algorithm, the row is of size  $O(\sqrt{n})$  and so the runtime for each row is  $O(\sqrt{n})$ . The addition of each of the  $\sqrt{n}$  rows happens independently, in parallel, leading to a  $O(\sqrt{n})$  runtime for all rows. Next, we increment each solution in each row of the addition step, keeping both the new and old values. As with the first step, each row can be completed independently in parallel by way of a linear growth across the row leading to a total runtime of  $O(\sqrt{n})$  for this step. After we increment our current working solutions we must both generate and propagate the carries for each row. In our algorithm, this simply involves growing a line across the leftmost wall of the rows. The size of the wall is bounded by  $O(\sqrt{n})$  and so this step takes  $O(\sqrt{n})$  time. Finally, in order to output the result bits into their proper places we simply grow a line atop the line created by the increment step. This step has the same runtime properties as the addition and increment steps. Therefore, the output step has a runtime of  $O(\sqrt{n})$  to output all rows.

There are four steps each taking  $O(\sqrt{n})$  time leading to a total runtime of  $O(\sqrt{n})$  for this algorithm. This upper bound meets the lower bound presented in Corollary 3.5 and the algorithm is therefore optimal.

**Choice of  $\sqrt{n}$  rows of  $\sqrt{n}$  size.** The choice for dividing the bits up into a  $\sqrt{n} \times \sqrt{n}$  grid is straightforward. Imagine that instead of using  $O(\sqrt{n})$  bits per row, a much smaller growing function such as  $O(\log n)$  bits per row is used. Then, each row would finish in  $O(\log n)$  time. After each row finishes, we would have to propagate the carries. The length of the west wall would no longer be bound by the slow growing function  $O(\sqrt{n})$  but would now be bound by the much faster growing function  $O(\frac{n}{\log n})$ . Therefore, there is a distinct trade off between the time necessary to add each row and the time necessary to propagate the carry with this scheme. The runtime of this algorithm can be viewed as the  $\max(row\_size, westwall\_size)$ . The best way to minimize this function is to divide the rows such that we have the same number of rows as columns, i.e. the smallest partition into the smallest sets. The best way to partition the bits is therefore into  $\sqrt{n}$  rows of  $\sqrt{n}$  bits.

### 5.3 Correctness.

The first two steps of the algorithm are addition followed by incrementation. It is important to note that this incrementation step not only outputs both the original (addition result) and incremented value but also whether or not the original value contained a zero. This is important because it will allow us to later use this information to decide whether or not a row will contain a carry. Now, these two steps of the algorithm rely on nothing but the data in the current row and are completed in parallel across all rows. Therefore, with the given tile set in Figure 10, these two steps are straightforward to verify.

The next step is for every row to select a solution from the two that were generated as well as propagate its carry information. We will, for the moment, assume that some row has not received the information of the previous carry and will concentrate on this row. At this step we know if the current row generated a carry  $C \in \{T, F\}$ , if the rows sum contains a zero  $Z \in \{T, F\}$ , and two possible row values  $A$  and  $B$ .  $A$  represents the sum if the row receives a carry-in of 0 and  $B$  the opposite. In order to continue from this step an answer must be selected and a carry or no-carry must be propagated. If  $Z = T$ , then we immediately know that we will propagate whatever  $C$  may be. If  $Z = F$  and  $C = T$ , we also know that we must propagate a carry. The only situation in which we do not know whether we will propagate a carry is when  $Z = F$  and  $C = F$ . When we encounter this situation we propagate whatever the carry was in the previous row. Also, in order to decide whether we will select  $A$  or  $B$  we need only the previous carry. Therefore, assuming we have the correct previous carry we can correctly select both the proper value for this row as well as propagate the correct carry to the next row.

Finally, because we know the initial carry is correct (it is part of the seed), we know that the first row can select the correct result as well as propagate the correct carry. Then, because we know that the first

row's carry is correct, we know that the second row can select the correct result as well as propagate the carry. This chain continues until it reaches the last row leading to selecting all the correct values as well as propagating all the correct carries.

The last step is to select the most significant bits value which is solely based on the last row's carry. If the last row propagates a carry it is a 1, otherwise it is a 0. Since we know that the last carry is correct we know that the last value is selected properly.

## 6 Towards Faster Addition

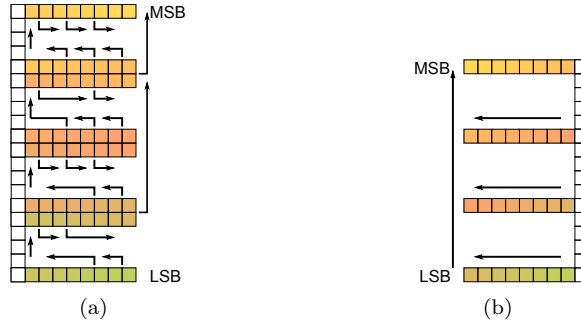


Figure 15: Arrows represent carry origination and direction of propagation for a) the  $O(\log n)$  average case, the  $O(\sqrt{n})$  worst case combined construction, and b) the  $O(\sqrt{n})$  worst case construction.

In this section we combine the approaches described in Sections 4 and 5 in order to achieve both  $O(\log n)$  average case addition and  $O(\sqrt{n})$  worst case addition. This construction resembles the construction described in Section 5 in that the numbers to be added are divided into sections and values are computed for both possible carry-in bit values. Additionally, the construction described here lowers the average case run time by utilizing the carry-skip mechanism described in Section 4 within each section and between sections.

**Theorem 6.1.** There exists a 2-dimensional  $n$ -bit adder TAC with an average run-time of  $O(\log n)$  and a worst case run-time of  $O(\sqrt{n})$ .

The proof follows from the construction of the adder in Sections 6.1, 6.2, and 6.3.

### 6.1 Construction

This construction combines the two-dimensional scaffold principle of the simple worst-case addition construction (Section 5) with the principle that certain addend-pairs can compute a carry out before they get a carry in, which was shown to reduce the average case run-time to  $O(\log n)$  in Section 4.2. Contrary to the addition construction in Section 5.1, the directionality of adjacent rows is antiparallel in the construction described here. Every odd row beginning with row one, which is the southernmost row, has the least significant bit on the east and the most significant bit on the west. Every even row has bits in the opposite order, as shown in Figure 18a. Each odd row, along with the even row above, should be considered as a single section in which the addition mechanism is nearly identical to the  $O(\log n)$  average case adder TAC. Within each section, carry outs are propagated east to west on the odd row, up in constant time from the most significant bit on the odd row (OMSB) to the least significant bit on the even row (ELSB), and from west to east on the even row. Adjacent rows are anti-parallel so that the distance between the most significant bit (MSB) of each section is a constant distance from the least significant bit (LSB) of the section above. This modification allows us to apply the  $O(\log n)$  average case addition between each pair of bits, and at the same time apply the carry propagation mechanism of the  $O(\sqrt{n})$  worst case addition construction between the MSB of each even row.

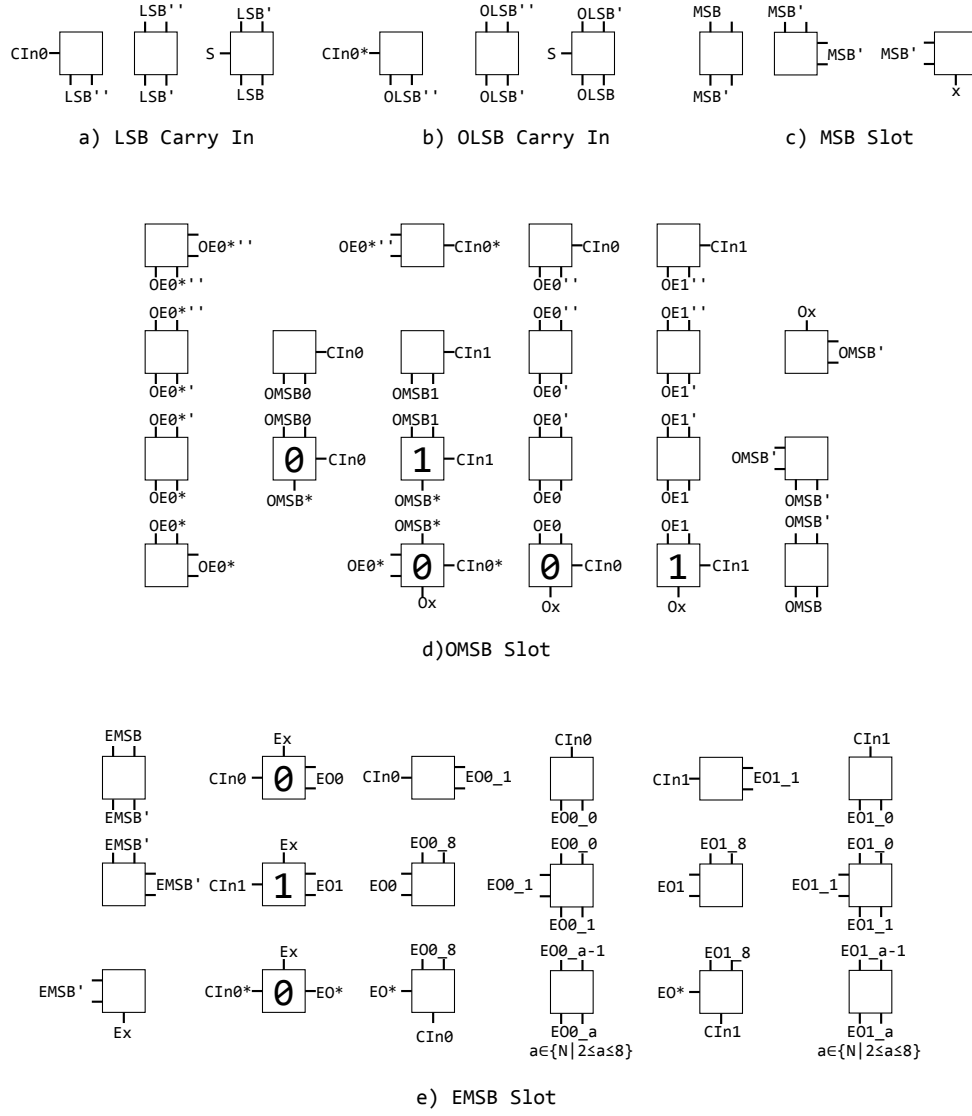
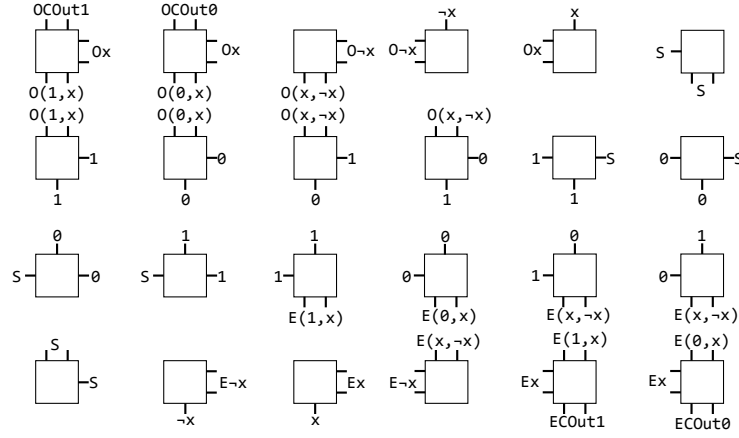
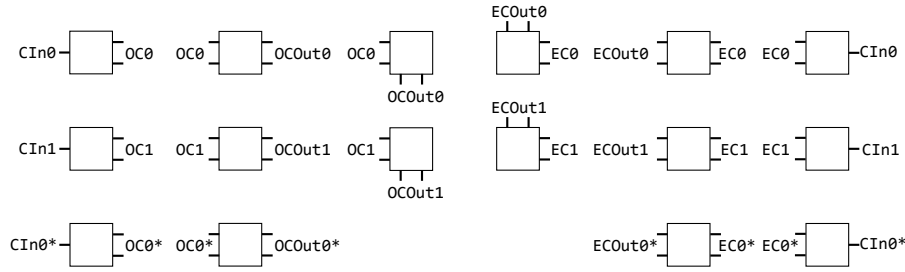


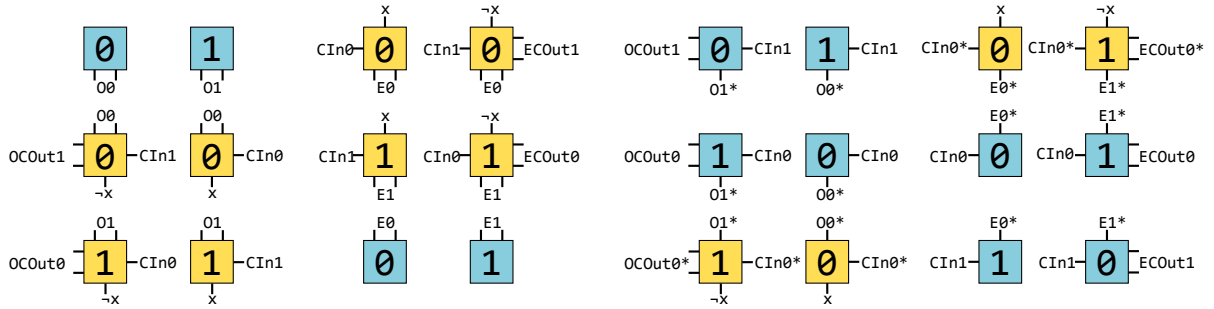
Figure 16: Partial set of tiles necessary to implement  $O(\log n)$  average case,  $\sqrt{n}$  worst case combined addition (see also next figure).



a) Addition



b) Carry Transfer



c) Print Answer

Figure 17: Tiles necessary to implement  $O(\log n)$  average case,  $\sqrt{n}$  worst case combined addition (continued from previous figure).

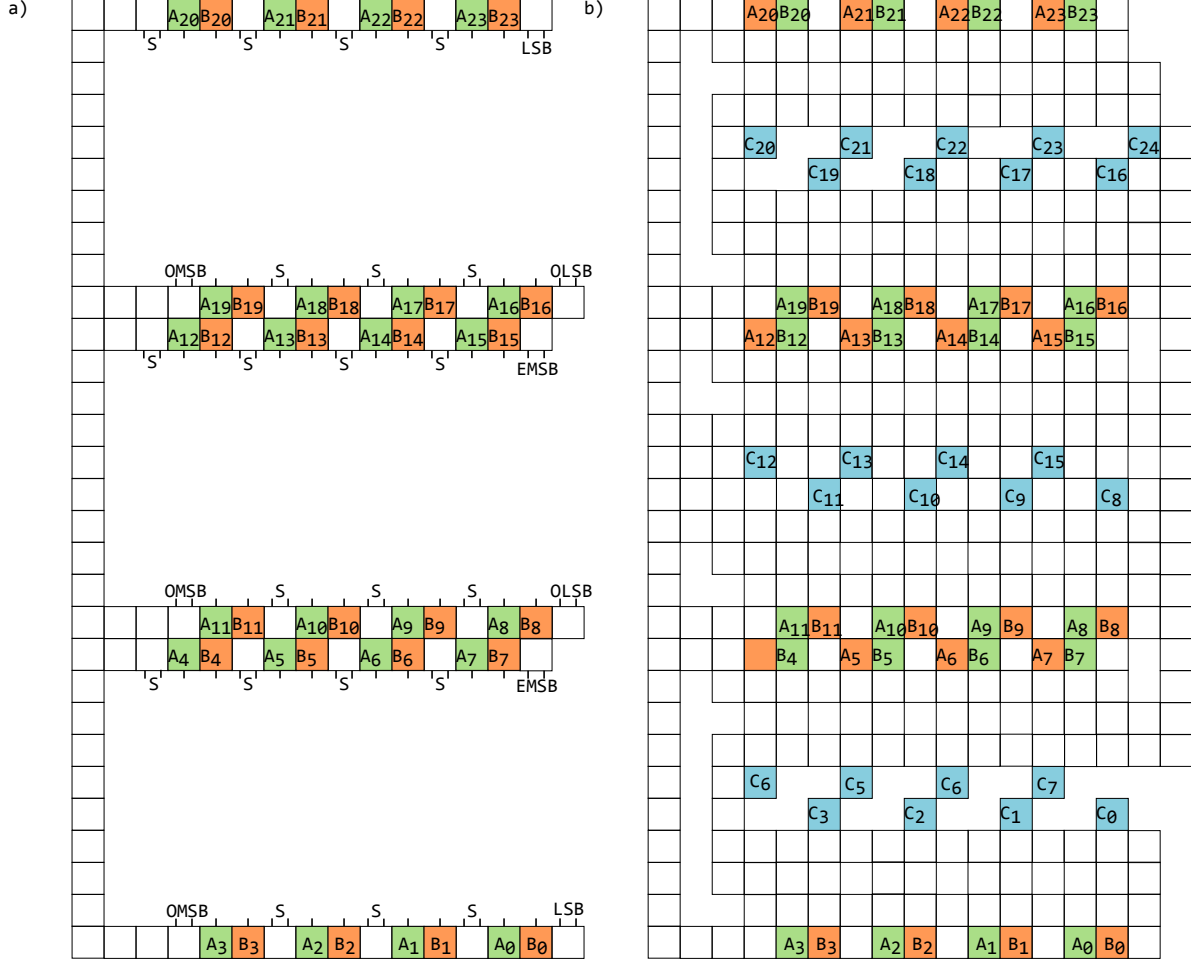


Figure 18: a) The input template for addition of two  $n$ -bit binary numbers  $A$  and  $B$ . b) The output template for the addition of two  $n$ -bit binary numbers  $A$  and  $B$ .

**Carry Passing and Prediction of Results Within Sections.** Within each section, or odd/even row pair, carry outs are propagated as stated in the above paragraph (Figure 19a). In addition, each row will present “predicted” results, that is, the result of the computation as if there were no carry from the MSB of the previous section. These results are shown as yellow tiles in Figure 19b-c. Note the blue tiles in Figure 19d. These blue tiles represent the final sum of the addend pairs. They may be computed without carry information propagated from a section below because the addend pairs are either 1) located in the southernmost section, where it is immediately known that the carry-in is 0 or 2) are flanked by a less significant addend pair for which the carry out is immediately known. The north face glues on yellow tiles in Figure 19d which contain a star \* rely on a carry from a previous section. The mechanism of carry propagation within a section can be seen in Figure 20a-b. Vertical columns grow up along the west side of each section to propagate the carry from the odd row to the even row.

**Carry Propagation Between Sections.** Figure 20c shows a vertical column growing between the southernmost section and the section above. This column is propagating a carry from the most significant bit of the southernmost section  $A$  to the least significant bit of the section  $B$  to the north. This information continues to the most significant bit of section  $B$  at which point a carry bit is computed to propagate to the

section north of  $B$  (Figure 20d, Figure 21a). The terminal tile assembly with the computed sum is presented in Figure 21b.

## 6.2 Time Complexity.

**$O(\sqrt{n})$  Worst Case and  $O(\log n)$  Average Case Run-Time** The construction presented in this section represents a combination of the constructions presented in Sections 4 and 5. Thus, this runtime analysis combines elements from the time complexity proofs in 4.2 and 5.2. In the construction described here, each section on the scaffold precomputes the sum as if there were a carry from the previous section and also as if there were no carry from the previous section. In some cases, whether a carry bit is passed in from the section below is irrelevant and parts of the final sum may be generated before this information is available. If the most significant addend-pair bits (MSB) of a given section can immediately generate a carry out for propagation to the next section, they do so. When a carry in arrives from the previous section, final values are selected from the precomputation step, if necessary. In this analysis of time complexity, we first analyze the run time of the precomputation step, and then consider the run time of propagating carry bits from section to section up the scaffold.

Consider the binary sequence  $T$  of length  $2n$  composed of  $n$ -bit binary numbers  $A$  and  $B$  as defined in Section 4.2. For this construction, divide  $T$  into  $\frac{\sqrt{n}}{2}$  smaller sequences, or sections,  $S_0, S_1, \dots, S_{\frac{\sqrt{n}}{2}-1}$ , with each section containing  $2\sqrt{n}$  addend-pairs. These sections are arranged on a scaffold as depicted in Figure 22. Note that the distance between the bottom half and the top half of a given section is constant and requires a constant number of steps to traverse. We first treat each section,  $S_i$ , as an independent addition problem without regard for a carry in from the previous section. Define  $k$  as the longest contiguous sequence of  $(0, 1)$  and  $(1, 0)$  addend-pairs in  $S_i$ . It follows from the proof in Section 4.2 that the run-time for  $S_i$  is bounded upwards by the length of  $k$ . The worst-case runtime for addition over the sequence  $S_i$  would thus occur when  $k = 2\sqrt{n}$ . Therefore, the worst case run time for addition *within* a section is  $O(\sqrt{n})$ . It also follows, as described in Section 4.2, that the expected length of  $k$  is  $O(\log n)$ . Therefore, the average run time for each addition within each section is  $O(\log n)$ . The precomputation within each of the  $\frac{\sqrt{n}}{2}$  sections occurs independently in parallel, leading to a worst case  $O(\sqrt{n})$ , average case  $O(\log n)$  precomputation run-time for all sections.

After performing this precomputation within each section, we must propagate carries between each section. The distance between two neighboring sections is constant and may be traversed by a column of tiles in a constant number of steps. Therefore, a carry out bit, once generated, can be propagated from one section to the next in constant time. A carry out bit may be propagated to the next section,  $S_{i+1}$ , immediately if the most significant addend pair of section  $S_i$  consists of  $(0, 0)$  or  $(0, 1)$ . Otherwise, this most significant addend pair of  $S_i$  must wait for a carry in before generating a carry out. Therefore, the composition of the most significant addend-pairs of the sections acts as a limiting factor to the speed with which carries may be propagated across all of the sections. Consider the binary sequence  $P$  which is comprised of the most significant addend pairs of each section and has a length of  $\frac{\sqrt{n}}{2}$  addend-pairs. Let  $k$  be the longest contiguous sequence of  $(1, 0)$  and  $(0, 1)$  addend-pairs in  $P$ . The propagation of carry bits through  $P$  is bounded upwards by the length of  $k$ , with the worst-case being when  $k = \frac{\sqrt{n}}{2}$  and an average case being when  $k = O(\log n)$ . Thus, the propagation of carry bits between each section up to the most significant addend-pair of  $T$  is bounded upwards by  $O(\sqrt{n})$  and has an average run-time of  $O(\log n)$ . Therefore, this addition algorithm has an upper bound of  $O(\sqrt{n})$  and an average run-time of  $O(\log n)$ .

## 6.3 Correctness.

Every row with north facing glues performs its addition in the exact same way as described in Section 4.1 assuming an incoming carry of 0. As such we use the proof of correctness from Section 4.3 to show that this step is correct. Also, every row with south facing glues performs this same addition except with an incoming carry from the addition of the lower north facing glue side. Thus, the same proof also applies to this side.

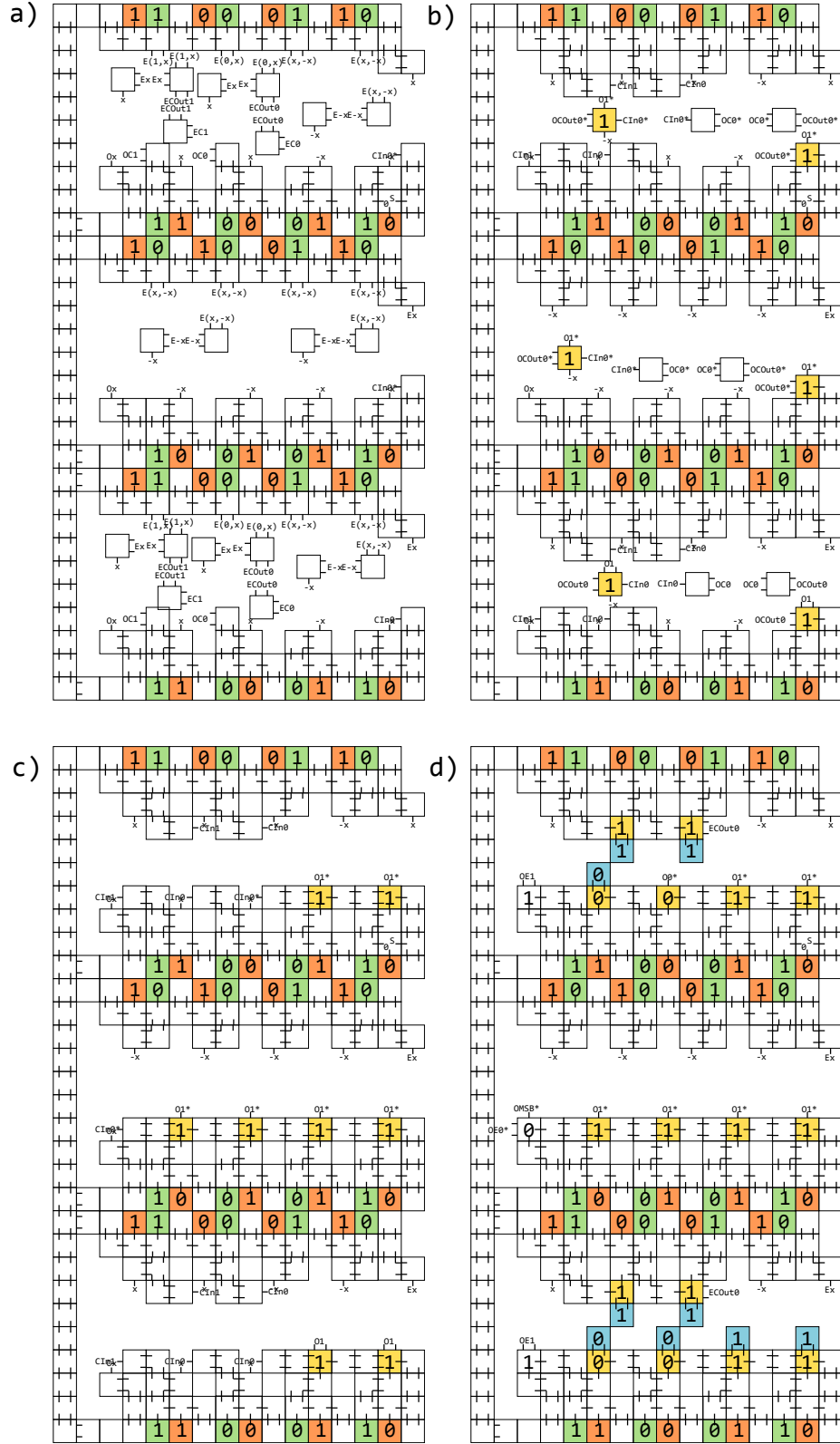


Figure 19: a) Carry out bits are computed for each addend pair where a carry out can be deduced immediately. b-c) "Predicted" results (yellow tiles) begin to attach. d) Blue tiles represent final result.

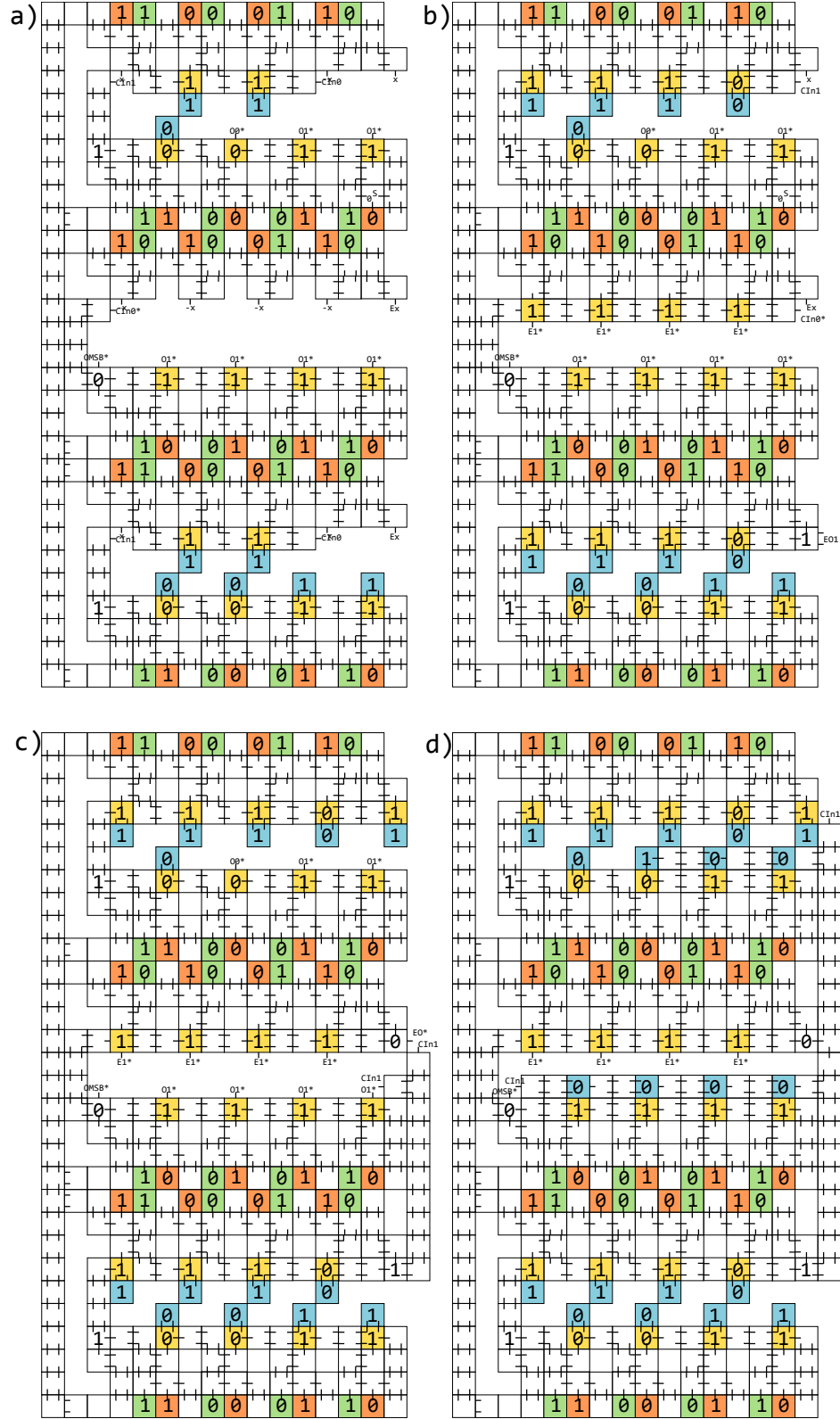


Figure 20: a-b) Carry bits are propagated within sections from the MSB of the odd row to the LSB of the even row using vertical columns. c-d) Carry bits are propagated between sections via vertical columns on the east side of the assembly.



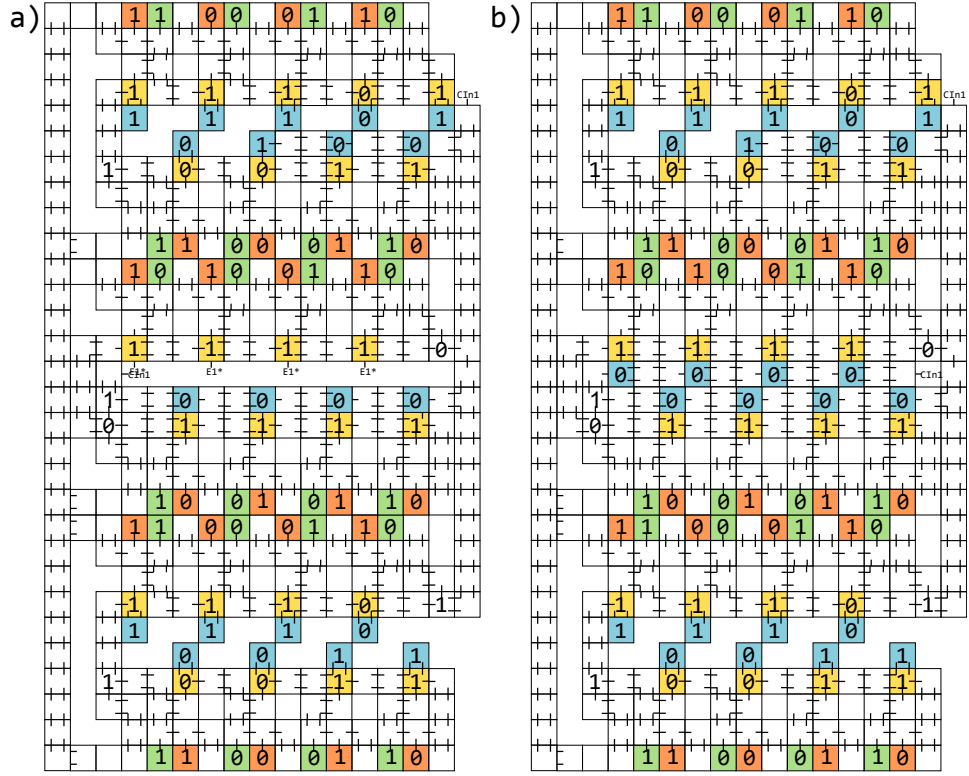


Figure 21: a) A carry bit from the first section propagates through the middle section in this assembly. b) The terminal assembly displaying the output  $C$ .

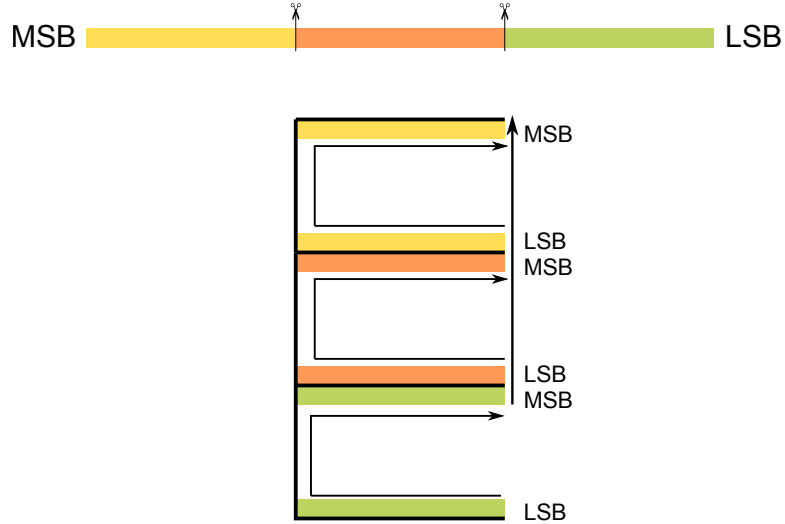


Figure 22: Top:  $T$  is separated into sections. Bottom: Each section of  $T$  is arranged on a scaffold. Arrows represent the general direction of carry bit propagation. MSB denotes the most significant bit of a section, and LSB denotes the least significant bit of a section.

Considering both of these additions as the first step, we can say that the first step correctly adds a section with a carry-in of 0.

Since the first section has no sections before it, the addition of the first section has an input carry of 0 allowing the copying of its value up to the final “display” row. Once an addend-pair pair knows its carry-in it can display its results. This combined with the fact that we proved the addition of any section with an input carry of 0 proves the correctness for the first section.

Once a section finishes the first step, there are three possible carry-outs from the MSB addend-pair in the section: 0, 1, and 0\*. If the section’s MSB addend-pair propagates a 0 or 1 then somewhere in the section some addend-pair generated its carry without depending on a carry from the previous section leading to a correct propagation. If the section’s MSB addend-pair propagates a 0\*, then no addend-pair in the section was able to generate a carry, meaning that no addend-pair in the section contained equal bits. Therefore, the section will correctly propagate whatever carry it received to the next section. These two clauses cover all cases in terms of carry propagation from one section to next section.

Every section other than the first one performs its addition with an input carry of 0\* which indicates an unknown carry with a value of 0. In other words, it continues the calculation assuming a 0 input carry but cannot copy any undetermined values, due to the unknown carry, into the “display” row until it receives the real carry from the previous section. This carry only gets propagated until it reaches an addend-pair with equal bits because at this point the addend-pair would have already generated its carry regardless of any incoming input carry and propagated it. One can see that the beginning of any section, other than the first, is essentially calculated by the algorithm as if it was in the center of some series of addend-pairs with unequal bits. When the correct carry propagates from the previous section to the current section, the correct values may then be copied into the “display” row. If a 0 is carried in, then it is a simple copy up of the value into the “display” row. If a 1 is carried in, then it is the inverse of what was previously calculated. Assuming some section receives the correct carry from the previous section, that section will “display” the correct result. We have shown that the first section propagates a carry correctly, and therefore all subsequent sections propagate their correct carry out. We have also shown that if each section propagates the correct carry-out to the next section, then the addition of addend-pairs within each section is performed correctly, producing the correct sum of  $A$  and  $B$ ,  $C$ .

## 7 Simulation

Tile self-assembly software simulations were conducted to visualize the diverse approaches to fast arithmetic presented in this paper, as well as to compare them to previous work. The adder tile constructions described in Sections 4-6, and the previous best [4] were simulated using the two timing models described in Sections 2.3.

Figure 23 demonstrates the power of the approaches described in this paper compared to previous work.

## 8 An Extension Towards 3-Dimensions

In this section we extend our construction into the third dimension to achieve a  $O(\sqrt[3]{n})$  upper bound, which meets the  $\Omega(\sqrt[3]{n})$  lower bound from Theorem 3.4. Due to space and time constraints a general overview is given. We begin by creating  $\sqrt[3]{n}$  total  $\sqrt[3]{n} \times \sqrt[3]{n}$  constructions exactly as per Section 6 stacked one atop the other in an alternating fashion such that every odd plate, beginning with the first, has its MSB in the NW corner while the even plate has its LSB in that same corner. We continue by applying the same addition algorithm that was presented in Section 6 to all plates where every lower plate passes its carry-out to the upper plate. This is very similar to how carry out bits are passed between sections in the construction described in Section 6. Finally, every lower plate will pass the appropriate carry to the next lower plate. Please see Figure 8 for a visual overview of this process.

**Theorem 8.1.** There exists a 3-dimensional  $n$ -bit adder TAC with an average run-time of  $O(\log n)$  and a worst case run-time of  $O(\sqrt[3]{n})$ .

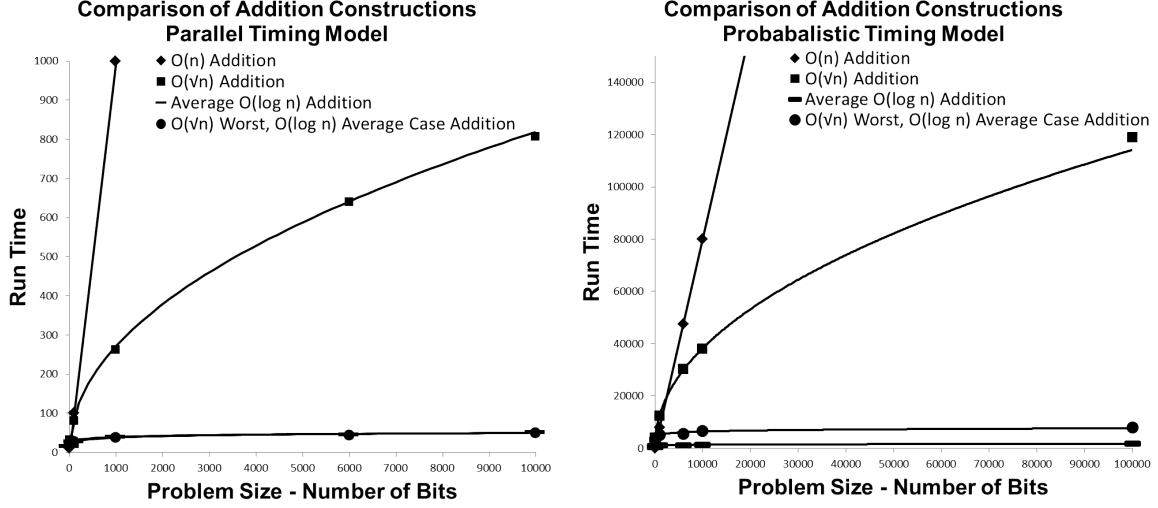


Figure 23:

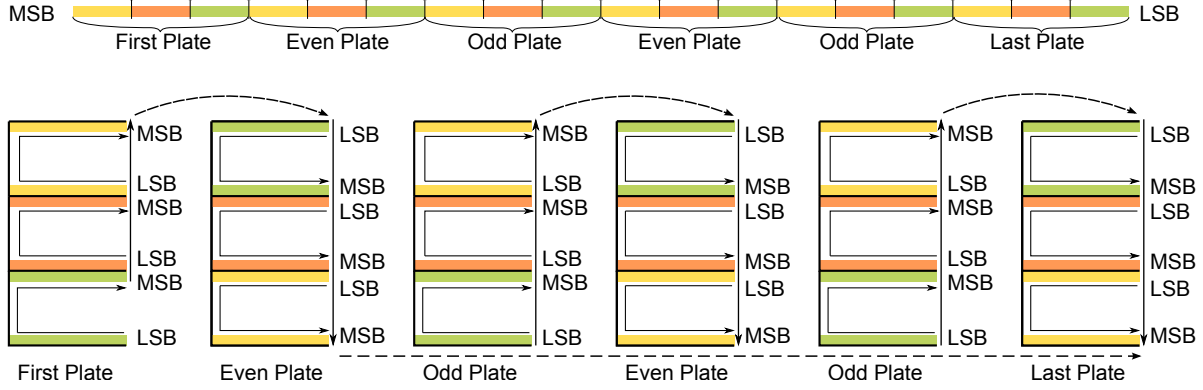


Figure 24: An overview of the process to add two number in optimal time in  $3D$ .

We present a high-level sketch of the proof here. Similar with the  $O(\log n)$  average case  $O(\sqrt{n})$  worst case combined addition presented in Section 6, the binary numbers to be added are separated into sections each with length  $O(\sqrt[3]{n})$ . We arrange these numbers on  $\sqrt[3]{n}$  scaffolds of size  $\sqrt[3]{n} \times \sqrt[3]{n} \times 2D$ . The dashed lines in Figure 8 are carries transmitted between each plate in the third dimension. The lines on the north are carries transmitted from each odd plate to its next even plate. The dashed lines on the southernmost part of Figure 8 are the carries transmitted from the first grouped plates to the last plate. Since the numbers are all fit compactly with only a constant amount space between each plate and a constant amount of space between each section, any one side of the cube is at most  $O(\sqrt[3]{n})$ . Therefore, this size constraint along with the algorithms previously presented allow us to have an optimal  $O(\sqrt[3]{n})$  time complexity with an average case complexity of  $O(\log n)$ .

## 9 Future Work

The results of this paper provide numerous directions for future work. One interesting open problem is whether it is possible to achieve  $n$ -bit multiplication in sublinear time, and if possible, how close can we get to the known lower bound of  $\Omega(\sqrt[3]{n})$  in dimension  $d$ . We conjecture that sublinear multiplication is possible,

especially if 3D systems are considered. If restricted to two dimensions, either a sublinear result or a proof of a  $\Omega(n)$  lower bound would be very interesting.

Another direction is the study of the difference between the parallel run time model and the probabilistic run time model. With tile sets of constant size, can it be shown that probabilistic run time is at most a logarithmic factor slower in the size of the seed assembly? Can a better bound be proven? What connection can be made for non-constant sized tile sets? Some of these questions have been considered in [3] for systems seeded with a single tile.

A final direction focusses on the consideration of non-deterministic tile assembly systems to improve expected run times even for maniacally designed worst case input strings. Is it possible to achieve  $O(\log n)$  expected run time for the addition problem regardless of the input bits? If not, are there other problems for which there is a provable gap in achievable assembly time between deterministic and non-deterministic systems?

## Acknowledgements

We would like to thank Damien Woods and Florent Becker for discussions regarding timing models in self-assembly. We would also like to thank Matt Patitz for helpful discussions of tile assembly simulation.

## References

- [1] Zachary Abel, Nadia Benbernou, Mirela Damian, Erik Demaine, Martin Demaine, Robin Flatland, Scott Kominers, and Robert Schweller, *Shape replication through self-assembly and RNase enzymes*, SODA 2010: Proceedings of the Twenty-first Annual ACM-SIAM Symposium on Discrete Algorithms (Austin, Texas), Society for Industrial and Applied Mathematics, 2010.
- [2] Leonard Adleman, Qi Cheng, Ashish Goel, and Ming-Deh Huang, *Running time and program size for self-assembled squares*, Proceedings of the thirty-third annual ACM Symposium on Theory of Computing (New York, NY, USA), ACM, 2001, pp. 740–748.
- [3] Florent Becker, Ivan Rapaport, and Eric Rémila, *Self-assembling classes of shapes with a minimum number of tiles, and in optimal time*, Foundations of Software Technology and Theoretical Computer Science (FSTTCS), 2006, pp. 45–56.
- [4] Yuriy Brun, *Arithmetic computation in the tile assembly model: Addition and multiplication*, Theoretical Computer Science **378** (2007), 17–31.
- [5] Nathaniel Bryans, Ehsan Chiniforooshan, David Doty, Lila Kari, and Shinnosuke Seki, *The power of nondeterminism in self-assembly*, SODA 2011: Proceedings of the 22nd Annual ACM-SIAM Symposium on Discrete Algorithms, SIAM, 2011, pp. 590–602.
- [6] Harish Chandran, Nikhil Gopalkrishnan, and John H. Reif, *The tile complexity of linear assemblies*, 36th International Colloquium on Automata, Languages and Programming, vol. 5555, 2009.
- [7] Matthew Cook, Yunhui Fu, and Robert T. Schweller, *Temperature 1 self-assembly: Deterministic assembly in 3d and probabilistic assembly in 2d*, Proceedings of the Twenty-Second Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2011 (Dana Randall, ed.), SIAM, 2011, pp. 570–589.
- [8] David Doty, *Randomized self-assembly for exact shapes*, SIAM Journal on Computing **39** (2010), no. 8, 3521–3552.
- [9] David Doty, Jack H. Lutz, Matthew J. Patitz, Robert Schweller, Scott M. Summers, and Damien Woods, *The tile assembly model is intrinsically universal*, FOCS 2012: Proceedings of the 53rd IEEE Conference on Foundations of Computer Science, 2012.

- [10] David Doty, Matthew J. Patitz, Dustin Reishus, Robert T. Schweller, and Scott M. Summers, *Strong fault-tolerance for self-assembly with fuzzy temperature*, Proceedings of the 51st Annual IEEE Symposium on Foundations of Computer Science (FOCS 2010), 2010, pp. 417–426.
- [11] Bin Fu, Matthew J. Patitz, Robert Schweller, and Robert Sheline, *Self-assembly with geometric tiles*, ICALP 2012: Proceedings of the 39th International Colloquium on Automata, Languages and Programming (Warwick, UK), 2012.
- [12] Ming-Yang Kao and Robert T. Schweller, *Randomized self-assembly for approximate shapes*, International Colloquium on Automata, Languages, and Programming, Lecture Notes in Computer Science, vol. 5125, Springer, 2008, pp. 370–384.
- [13] Mark Schilling, *The longest run of heads*, The College Mathematics Journal **21** (1990), no. 3, 196–207.
- [14] Robert Schweller and Michael Sherman, *Fuel efficient computation in passive self-assembly*, Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms, 2013.
- [15] Erik Winfree, *Algorithmic self-assembly of DNA*, Ph.D. thesis, California Institute of Technology, June 1998.